Master Machine Code on your Amstrad CPC 464 & 664

Jeff Naylor & Diane Rogers

Master Machine Code on your Amstrad CPC 464 & 664

Jeff Naylor & Diane Rogers

First published 1985 by:
Sunshine Books (an imprint of Scot Books Ltd.)
12–13 Little Newport Street
London WC2H 7PP

Copyright © Jeff Naylor and Diane Rogers, 1985

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording and/or otherwise, without the prior permission of the Publishers.

British Library Cataloguing in Publication Data
Naylor, Jeff
Master machine code on your Amstrad CPC 464 and 664
1. Amstrad Microcomputer—Programming
2. Machine codes (Electronic computers)
I. Title II. Rogers, Diane
001.64'24 QA76.8.A4

ISBN 0–946408–82–3

The cover shows a screen shot from *Sorcery*, by kind permission of Virgin Games Ltd.

Cover photograph by Ian Batchelor.
Typeset and printed in England by Nene Litho,
Irthlingborough, Northants.

Contents

	Page
1 The Z80	**1**
Binary — a CPU's numbering system	1
Data bus	1
Memory and the address bus	1
Hexadecimal numbering	2
Two's complement	2
The clock	3
External connections	3
The registers	3
Index registers	5
Special purpose registers	5
Instructions	6
Load instructions	7
The accumulator	10
Increment and decrement	10
Logical instructions	11
Testing	12
Rotate and shift	12
Binary coded decimal	13
More complex instructions	14
Interrupts	16
Two final commands	16
2 Writing Code	**17**
Subroutines	17
Loops	17
Planning ahead	18
An example routine — INPUT	18
Writing the source code	20
Assembling the object code	21

Loading the routine 22
Testing INPUT 23
Two things to avoid when writing machine code 23

3 The Hardware and Firmware **25**
Memory devices 25
The gate array 25
Video display 26
Input and output 26
Sound generation 26
The keyboard 27
Firmware 27
Key manager 28
Text VDU 29
Screen routines 29
Cassette manager routines 30
Sound 31
Kernel routines 31
The machine pack 31

4 Machine Code Number Sorting **33**
Sorting methods 33
The human approach 33
The computer approach 34
Integer array sorting 37
Using the integer sort demonstration 39
Handling signed numbers 40
Swapping the values 42
Sorting floating point arrays 43
Real sort demonstration 44
The format of real numbers 44
SortReal — a floating-point sorting routine 46
Comparing values 49
The SWAP? section 50

5 Sorting String Arrays **53**
String arrays in BASIC 53
SortString in action 55
SWAP? 56
SortArray — the heart of a database program 57
Database 62
Parameters for SortArray 64
SortArray in action 65
Comparison section 69
Comparing strings as numbers 69
The swap section 70

6 Enhancing Database **73**
A search routine 73
Implementing Search 74
The ArrayShift routine 82
Pulling yourself up by your bootstraps 86

7 Graphics — a Circle Routine **91**
Circle machine code in detail 97
Multiplying in binary 98
Calculating the co-ordinates 99
Drawing the circle 99
Estimating square roots 100
Binary searching 101
SQRLP — machine code to find a square root 101
Graphics firmware routines 102

8 A Fill Routine **105**
Using Fill 107
Screen co-ordinate systems 107
The screen memory layout 108
The arrangement of the screen memory map 109
The initial video RAM layout 110
Changing the base and offset of screen RAM 111
Masking — how we isolate individual pixels 111

The principles behind Fill 112
The line fill subroutine — LINFL *113*
The Fill listing *113*
Screen pack firmware routines *117*

9 Extending BASIC **119**
Setting up the RSX 120
RSX tables 122
The RSX routine 122
Error handling *124*
General purpose subroutines 124
Predictability *125*
Flexibility *125*
Delay *126*
Strings *127*

CHAPTER 1
The Z80

At the heart of every computer there is what is known as a Central Processing Unit. Microcomputers, as you might guess employ a microprocessor as their CPU. The purpose of the CPU is to translate programs into actions. The Amstrad employs a Z80 microprocessor, which is an eight-bit chip with 16-bit addressing. It runs at a clock speed of 4 Mhz. Don't worry if you do not understand these terms now, they will be explained later.

Binary — a CPU's numbering system

Humans normally represent numbers with symbols that can have values between zero and nine. Once a column reaches its capacity, this *decimal* notation uses further columns, each having ten times the value of the previous one to its right. Computers are limited to two symbols, 0 and 1, which are represented electrically by on or off. This numbering system is known as *binary*, with each column having twice the value of one to its right. Hence, 11111111 represents 1 plus 1*2 plus 1*4 plus 1*8 plus 1*16 plus 1*32 plus 1*64 plus 1*128, which works out as 255 decimal.
Using eight columns, or *bits*, a binary numbering system allows us, or a computer to represent values in the range 0 to 255 in decimal. This eight-bit number is known as a *byte*.

Data bus

The Z80 is an eight-bit CPU which handles data one byte at a time, so that its basic unit of data is limited to the range 0-255. Of course larger values can be handled but this involves using more than one byte, and so must be carried out sequentially. In order to communicate with other circuits there are eight lines from the CPU which transmit and receive data, known collectively as the *data bus*.

Memory and the address bus

Memory storage for values, data and programs is essential if a computer is to

function usefully. There are two basic types of memory, both of which are found in your Amstrad computer: ROM (Read Only Memory), from which the CPU can obtain the eight-bit values it requires, and RAM (Random Access Memory), which the CPU can use to store byte values, and retrieve them at some time in the future. Each memory location has its own *address*, so that it, and only it, responds when that address is sent out by the CPU. In order to generate these addresses, the Z80 is equipped with 16 pins, which make up the *address bus* and allow a 16-bit number to be created. Thus the normal limit to the number of memory locations that the Z80 can address is that which can be represnted by two consecutive bytes: 256*256 or 65536. one 'K' of memory bytes is not, as you might think, 1000, but the largest number that can be represented by 10 bits, 1024. This explains why you will read of the Z80 being capable of addressing 64K bytes of memory; 65336 is 1024*64.

Hexadecimal numbering

People use decimal numbers and computers use binary, but most writers of machine code programs also use *hexadecimal*. This numbering system works to a base value of 16, with five extra symbols, the letters A - F being used to represent the decimal values 10 to 15. The convenience of this system may not be obvious immediately, particularly as most people have trouble familiarising themselves with these new numbers, but in fact it enables machine code to be written in a very compact form — a sort of shorthand.
The advantage of the hexadecimal numbering system is that it bears a close relationship with binary, without being as cumbersome to write down: a single byte can be represented as two hex digits — FF equals 255; if you come across a number with three hex digits you know immediately that it will need up to 12 bits to represent it. Locomotive BASIC is equipped with HEX$ and BIN$ functions to allow you to convert numbers easily.

Two's complement

The convention of 'two's complement' arithmetic allows negative numbers to be represented. For the system to work, we must use the same size of binary number (i.e. same number of bits) calculations. The most significant bit is used to indicate the sign. In the case of eight-bit two's complement numbers, bit 7 is used to indicate whether a number is positive or negative and numbers can be expressed in the range −128 to +127. If bit 7 is zero, the number is positive and the low seven bits give it a value in the range 0 to 127. If bit 7 is 1, however, the number is negative and can be calculated in the following manner. Invert each bit of the number (this process is known as complementing), then subtract 1. The number remaining, with a minus sign in

front, is the value of the byte.
The usefulness of this convention can be seen when we add two numbers together. Adding a negative number to a positive one will result in the correct answer. A simple example is the addition of −1 (11111111 bin) to +1 (00000001). Add the two together and the eight-bit result is 00000000 bin — the 1 carried into the ninth bit is lost and the answer is zero.
Appendix II is a table of eight-bit two's complement values. The system is also used for 16-bit values; when FFFF hex is −1, for example.

The clock.

The speed at which the Z80 performs each step of its operations is dictated by the *clock*. Your Amstrad contains an oscillator circuit which sends pulses to the CPU; each pulse triggers the next stage in whatever function is currently being performed. For example, if the processor is attempting to fetch a byte from a memory location, it will send out the address and other signals, and then wait for another clock pulse before reading the data in. The speed of the clock allows other circuits such as RAM to keep up with the processor.

External connections

A Z80 processor has 40 terminals or pins. Some of these are used to feed the CPU the clock pulses and voltage supply (+5 volts) which brings it to life; other pins send and receive various controlling signals, such as read and write. The address and data buses are connected to the Z80 through two further sets of pins — D0 to D7 for the data bus and A0 to A15 for the address bus.
Internally, the Z80 can be divided into three areas — the *Control Unit*, the *Arithmetic and Logic Unit* (ALU) and a number of *registers*.
The control unit translates an instruction into the correct sequence of signals in order to perform the required task and it has associated with it a register, or local memory cell; this register holds the machine code instruction on which the control unit is currently working.
The ALU is capable of performing simple addition, subtraction and logic operations between two values sent to it along internal data paths.
The registers are a collection of local memory cells, each capable of storing eight-bit or 16-bit numbers. Their contents can be sent to the outside world, other registers or the ALU, and they can be loaded with data from the same places.

The registers

The most important of the registers is the *Program Counter*, or PC: this is a 16-bit register which is used to keep track of the program that the CPU is

running. The program information is fetched, one byte at a time, from memory, and the PC holds the address of the next byte which will need to be read in. Another 16-bit register with a special purpose is the *Stack Pointer*, or SP. This holds the address of an area of memory which is used as storage space by certain instructions. (See the POP, PUSH and CALL instructions for a description of its uses.) The main group of registers for use by the programmer, called the general purpose registers, falls into two sections. The Z80 has 14 eight-bit registers but at any one time only one set of seven registers can be used: these are called A (or the accumulator), B, C, D, E, H and L. The other seven registers share the same names, but they are known as the alternative register set. A pair of exchange instructions is all that is required to bring the alternative set of registers into operation. The resident operating system and BASIC language make extensive use of the alternative set of registers, so for most applications you will not want to have access to them.

If you do wish to make use of them, however, special arrangements must be made, even in pure machine code programs.

The accumulator, A, is the eight-bit register which is used for many of the arithmetical operations, and all the local ones. The remaining registers in the general purpose group can be used as single eight-bit storage registers, with the advantage of speed over using external memory locations. They may also be combined into 16-bit registers for the handling of address information and other double byte data. When paired off in such a manner, they are known as BC, DE and HL. In general the second register of a pair holds the least significant byte of data: if we stored 256 (100 hex) in the register pair HL, H would hold 1 (representing 256*1), and L zero. The letters of HL help us to remember this; the Z80 was developed from a chip, the 8080, in which H stood for high and L stood for low. A very special eight-bit register is sometimes paired with A: it is called F, or the *flags* register when combined the pair is called AF. Each individual bit of F is itself a flag, and is used by the ALU and control unit to make a note of particular ocurrences. Some instructions have no effect on the flags at all, others affect them to varying degrees. For example, after every ADD operation that is carried out, one bit of the F register, known as the C or CARRY flag, will be set to 1 if the addition resulted in an overflow; this will occur if the answer was too big to store in the register that was to receive the result.

The flags, in the order of their bit number in the F register, behave as follows:

Carry flag: used to store the most significant bit of arithmetic operations. Logic operations reset it to zero. It is also involved in some of the shift and rotate instructions.

Subtract flag: mainly for CPU rather than programmer use. Set each time a

subtract is carried out, reset by an add.
Parity/overflow flag: a dual purpose flag. After certain instructions, such as logic or rotate operations, it reflects the parity of the result. Parity is set if all the bits of the answer add up to an even value. This flag is used in a similar manner to the carry flag by certain block move and search operations.
Bit 3: this is undefined.
Auxiliary carry flag: reflects any carry from bit 3 to 4 during an arithmetic instruction. Its main purpose is for the handling of BCD (binary coded decimal) numbers.
Bit 5: also undefined.
Zero flag: set when any arithmetic or logic operation results in zero. It is cleared if the result is not zero.
Sign flag: after an arithmetic or logical operation it reflects the state of the most significant bit of the result.
These flags are very important to the programmer; they offer the opportunity to carry out conditional operations.
In the above description I have talked of the flags being set (value 1) or reset (value 0). It is worth mentioning other ways used to describe the state of the flags. The term 'clear' is often used to mean that a flag has a zero value. Alternatively, logic terms are sometimes used: 'true' implies that a flag is set and 'false' that it is reset. You will meet such phrases as 'if carry is clear' (meaning the carry flag has the value 0) and even 'when Z is true' (the zero flag has the value 1).
Note that there is also an alternative flags register. For the purposes of the exchange instructions, it is paired with the alternative A register.

Index registers

Two 16-bit registers (which do not have alternatives) are provided by the Z80 to allow an indexing facility. These registers, IX and IY, have a number of instructions which enable them to be used as memory pointers. When loaded with an address, they can have an *offset* (or *displacement*) in the range −128 to +127 added to them and the result is then used as an address to reference memory.

Special purpose registers

There are two eight-bit registers which you will rarely use. The I (or interrupt vector) register is for use with an interrupt mode not employed by your computer. Interrupts are discussed later in this chapter. The *Refresh* register is really part of the Z80 hardware. In an operation completely transparent to the programmer, the CPU puts out signals to refresh dynamic memory chips

after each fetch of an instruction. To which addresses it sends these signals depends on the contents of R, and this is continually being incremented by the control unit. The only use which a programmer can normally make of R is as a random number generator: it is unlikely that a program will run at a speed which is exactly related to the frequency of the update of R, so it can be used to generate random events.

Instructions

The instructions which the Z80 receives come from memory locations. When first switched on and reset (the application of a signal to one pin of the Z80 to bring it to life), the first thing that the control unit does is to fetch the first instruction from memory location zero. This takes the form of a byte which is placed in the instruction register and acted upon. It may be that further bytes are required to complete the instruction — Z80 machine codes can consist of between one and four bytes, which are fetched from consecutive memory locations. The PC is incremented by one after each fetch. When the first instruction has been obeyed, the next one is taken from the memory location of which the address is in the PC.

Before we look at the instructions that are available on a Z80 microprocessor, let me say a few words about the following section of the book. It lists all the types of instructions; operations that are not mentioned are therefore not possible. The shorthand description, or mnemonic, is sometimes given in a generalised form with the following letters

r	General purpose eight-bit register. (A, B, C, D, E, H and L.)
rp	16-bit register pair. Always BC, DE and HL. The text will indicate if AF, IX and IY are also included.
addr	A 16-bit value for use as an address.
dis	Eight-bit displacement value.
index	Means that an address is derived from the contents of either IX or IY plus a displacement.

The use of brackets around a register pair or value means that the address contained in the pair or given as data is to be used to fetch the actual value from. LD A,(HL) takes the value stored at the address held in HL and stores it in A. In this case it is said that HL 'points' to the data, or that HL is 'being used as a pointer'.

While the mnemonics are listed here, the operation codes (shortened to 'op codes') are given in Appendix 1; these are the actual values that when encountered as instructions by the CPU cause the effect described by the mnemonic.

Load instructions

Instructions fall into a number of categories. The most frequently used group are *loading* instructions; these never affect the flags.

LD r,r: load any general-purpose eight-bit register (including A) with the contents of any other.

LD r,data: load the byte from the next memory location into any general-purpose eight-bit register.

LD r,(HL): take the contents of the memory location of which the address is contained in HL and place them into a general-purpose eight-bit register.

LD r,(IX+dis): add IX and the displacement (which is taken as a signed number), and load the value stored at the resulting address into the general-purpose eight-bit register. The 'two's complement' convention is used for the displacement, or offset, so FF is −1, FE −2 and so on to 80 hex, which is −128.

The A register has additional op codes:

LD A,(rp): the address of the data to be loaded into A can be taken from BC or DE as well as from HL, IX+dis or IY+dis.

LD A,(addr): the address of the data is supplied as two additional bytes, low byte first, in the next two bytes of program memory: e.g. LD A,(2040) takes the form 3A 40 20. This type of addressing is known as DIRECT.

It is also possible to transfer data between A and the I or the R register with these instructions:

LD A,I — LD A,R — LD I,A — LD R,A.

The register pairs BC, DE, HL, IX and IY can be loaded in the following manner:

LD rp,data: two bytes of data, low byte first, are taken from the next two program memory locations.

LD rp,(addr): the address supplied is used in the same manner as LD A,(addr) to find the value to be loaded into the low half of the pair; the high register is loaded with the byte from (addr+1).

All the instructions which fetched a value from an address (but not program memory) — have their complements, these instructions load the memory location specified with the contents of a register or register pair:

LD (HL),r — LD (IX+dis),r — LD (IY+dis),r.

LD (rp),A — LD (addr),A.

LD (addr),rp.

It is also possible to load immediate data from program memory into a location of which the address is held by certain pairs:

LD (HL) data — LD (IX+dis),data — LD (IY+dis),data.

The final loading operation affects the stack pointer:

LD SP,HL — LD SP,IX — LD SP,IY

Note that loading HL, IX or IY with the contents of SP is not possible.

Exchange instructions

Other instructions which have similar loading effects are the exchange instructions, which swop over the values of two locations:

EX (SP),HL: swop the two bytes at (SP) and SP+1) with those in HL. Also possible are EX (SP),IX and EX (SP),IY.

EX DE,HL: swops the contents of DE and HL. The other exchange instructions bring the alternative register set into operation — EX2 AF,AF' for the AF pair, and EXX, for the remainder. No other exchange operations are possible.

Set and Reset

Two instructions load single bits into specified locations. SET loads a 1, RES (reset) loads a 0. Any bit can be specified, such as SET 4,B, forcing bit 4 of B to the value 1.

SET bit,r: any G.P. register.

RES bit,r: any G.P. register.

SET bit,(HL), SET bit,(IX+dis), SET bit,(IY+dis): these set the specified bit in the memory location pointed to by HL or the index registers plus displacement.

RES bit,(HL), RES bit,(IX+dis), RES bit,(IY+dis): as above but they clear the bit to zero.

Another instruction which can be considered as a loading code is SCF. This sets the carry flag to 1.

Many other instructions have elements of loading involved.

Stack operations load and retrieve data and addresses from an area of RAM pointed to by the SP register. A stack is a LIFO storage method, meaning Last In First Out, and it is worth repeating an often used analogy. Imagine placing playing cards in a pile, or stack, on the table. You can only place cards on the top of the pile, and only retrieve cards by picking up the top card first. Therefore, if you put down the ace of spades and then three more cards, you must remove those three before you can rescue the ace.

Data must come from a 16-bit source, and is placed on the stack low byte first; the SP always points to the current 'top' of the stack. The one complication with Z80 stack operations is that they work from high to low addresses; as more data is put in the stack area, the SP points to lower addresses. Placing and retrieving data from the stack uses instructions called PUSH and POP.

PUSH rp: SP is decremented and the contents of the high half of the pair placed at the resulting address. SP is decremented again, and the low half of the pair's contents saved at this new address. A can be pushed onto the stack in conjunction with F — PUSH AF.

POP rp: the reverse process to PUSH, so that POP retrieves the information

which was last PUSHed. The value pointed to by SP is placed into the low half of the pair: SP is incremented, and the new value now pointed to is moved to the high half of the pair. Finally, SP is incremented again.

Jumps

We can load the PC with a new value; this has a major effect on the running of a program. By loading the address of another section of code into PC we can cause a diversion in the normally linear flow of a program. For this reason, loading the PC is given the name JUMP:

JP addr: the new address from which to continue execution is given in the next two program bytes (low byte first)

JP (HL) — JP (IX) — JP (IY) — the contents of the specified rp are loaded into the PC.

JP addr has a number of variants which test one of the flags before performing the load: if the flag tested does not meet the condition required then the load does not take place. These variants allow the flow of the program to be affected on a conditional basis, in much the same way as IF A=0 THEN GOTO 100 would do in BASIC. The possibilities are:

JP NZ: jump if the zero flag is 0.

JP Z: jump if the zero flag is 1.

JP NC, JP C: jump if the carry flag is 0 or 1 respectively.

JP PO, JP PE: jump if the parity flag is 0 or 1 respectively.

JP P, JP M: jump if the sign flag is positive or negative.

While JP needs a 16-bit address, another instruction, *jump relative*, adds an eight-bit offset to the value already in the PC. This means we do not need to know the address of the location to which we wish to jump, just how many bytes it is from the address already in the PC. Jumps to an address lower than the current value of the PC are achieved with negative offsets, in the same manner as those used in the indexing instruction. JR is only possible over a range −128 to +127.

JR dis: jumps the specified number of program bytes. Note that JR FE (18 FE) will form an endless loop, because FE is −2 in two's complement.

Conditional JRs are also possible, but with a limited number of tests:

JR NZ, JR Z — tests the zero flag.

JR NC, JR C — tests the carry flag.

One sophisticated jumping instruction allows program loops to be performed quickly:

DJNZ dis: decrement the B register, and if it does not reach zero, add the displacement to the PC.

CALL addr: similar to the BASIC GOSUB command. The current value in

the PC is pushed onto the stack and the PC loaded with the supplied address. Conditional CALLs are possible:
CALL Z, CALL NZ, CALL C, CALL NC, CALL PE, CALL PO, CALL N and CALL M.
RET: used at the end of a subroutine, it POPs the address onto the stack and puts it into the PC.
Returns may also be conditional, with the same conditions as CALL.

The Accumulator

The accumulator, A, is the register which is used to receive the results of eight-bit arithmetic.
ADD A,data: the data immediately following in program memory is added to A.
ADD A,r: the contents of the specified G.P. register are added to A.
ADD A,(HL): HL contains the address of the byte to be added to A.
ADD A,(IX+dis), ADD A,(IY+dis): versions of ADD using the index registers.

Add with carry performs in the same manner as ADD, but includes the carry bit in the sum:
ADC A,data, ADC A,r, ADC A,(HL), ADC A,(index).

Subtract is only available as an eight-bit operation on A.
SUB data, SUB r, SUB (HL), SUB (index).
Subtract with carry includes the carry bit in the calculation:
SBC data, SBC r, SBC (HL), SBC (index).
A limited number of 16-bit operations are possible, using HL as the recipient of the result:
ADD HL,rp: rp may be BC, DE, HL or SP.
ADC HL,rp: rp as ADD.
SBC HL,rp: rp as ADD.

It is also possible to ADD to the index registers:
ADD IX,rp: rp may be BC, DE, IX or SP.
ADD IY,rp: rp may be BC, DE, IY OR SP.
All of the above arithmetic operations have an effect on all of the flags.

Increment and decrement

A task that is very common in machine code programs is the addition or subtraction of 1 from a value. To facilitate this there are some special

operations:
DEC r: subract 1 from the specified eight-bit G.P. register.
INC r: add 1 to the specified eight-bit G.P. register.
DEC (HL): decrement the contents of memory location (HL).
DEC (index): index register versions of DEC (HL).
INC (HL): increment the contents of memory location (HL).
INC (index): index register version of INC (HL).
These instructions affect all the flags *except the carry*.
16-bit versions of the above are available:
INC rp, DEC rp — rp may be BC, DE, HL, IX, IY or SP.
NOTE — These do *not* affect any flags.

Logical instructions

The next group of instruction are known as *logical*, or *Boolean* operations.
AND is a logical operation that obeys the following rule: if the first bit AND the second bit are 1, then the result is 1. Otherwise, the result is 0. The operation is carried out bit by bit between the A register and the specified byte, with resulting bits placed in A.
AND data: AND the byte in the following byte of program memory with A, placing the result in A.
AND r: any G.P. can be used instead of immediate data.
AND (HL): the byte is fetched from location (HL).
AND (index): indexed version of AND (HL).

OR is a similar logical operation, with the following rule: if either the bit being compared in the accumulator or in the specified byte is 1, then the result is 1. If both bits are zero, the result is zero.
OR data, OR r, OR (HL), OR (index).

Exclusive OR (XOR) is a little more complex: if both bits are the same (either both 1 or both 0) then the result is 0, otherwise the result is 1. XOR can be thought of as 'not the same':
XOR data, XOR r, XOR (HL), XOR (index)
With logical operations the carry flag is always cleared, the auxiliary carry flag set, and the other flags are affected as normal.
Two instructions which perform operations on the A register without the need for another byte are:
CPL — complement A. All the bits are inverted. The auxiliary carry is set, the other flags left alone.
NEG — the same as subtracting the contents of A from zero and placing the result back into A. All flags are altered.

The carry flag has its own logic operation:
CCF — complement the carry flag.

Testing

Testing can be thought of as a subdivision of logic — 'If A is the same as the byte then set Z'.
CP r: the contents of the specified register are subtracted from the contents of A. The result is discarded, but the flags are set according to the result.
CP (HL): HL provides the address of the byte to be compared with A.
CP (index): indexed addressing version of CP (HL).
For testing individual bits of a target byte we can use BIT:
BIT bit,r: place the complement of the specified bit from the specified register into the Z flag.
BIT bit,(HL): HL provides the address of the byte containing the bit to be tested.
BIT bit,(index): indexed version of BIT (HL).
The S and P/O flags act in an unpredictable way; auxiliary carry is set and carry is unaffected.

Rotate and shift

There is a group of instructions which have a surprising number of uses; their task is to rotate and shift the bits in an eight-bit value:
RR r (Rotate Right): the rightmost bit (bit 0) is placed in the carry. Bit 1 is transferred to bit 0, 2 to 1 and so on. The old value of the carry is put into bit 7. This can be performed on any G.P. register.
RR (HL), RR (index): versions of RR that operate on a byte in memory of which the address is specified by HL or the index registers plus displacement.
RRA: a rotate right instruction specifically for the A register. It is only one byte long, and has a different effect on the flags.
RRC R (Rotate Right Circular): similar to RR, but the original value of bit 0 is placed into bit 7 instead of the old carry value. The carry flag also takes on the value of the original bit 0.
RRC (HL), RRC (index): memory versions of RRC.
RRCA: accumulator only RRC instruction.
All the above rotate instructions can be performed in the opposite direction; bit 7 into the carry, 6 into 7 and so on. These are Rotate Left instructions:
RL r, RL (HL), RL (index), RLA.
RLC r, RLC (HL), RLC (index), RLCA.

Shift instructions are a variant of rotate:

SRA r (Shift Right Arithmetic): bit 0 is placed into carry, 1 into 0, etc. The value in bit 7 is transferred to bit 6 but also retained in bit 7.
SRA (HL), SRA (index): memory versions.
SLA r: bit 7 is placed into carry, the others move up a place, and bit 0 is set to zero.
SLA (HL), SLA (index): memory versions.
SRL r (Shift Right Logical): as SRA, but bit 7 is replaced with a zero.
SRL (HL), SRL (index): memory versions.
The manner in which most of the rotate and shift instructions affect the flags is as follows:
Carry is determined by whatever is moved into it: auxiliary carry is cleared. The other flags are all determined according to the result of the operation.
The accumulator-only codes, RRA, RLA, RRCA and RLCA have no affect on most of the flags; carry is set by the bit moved into it and auxiliary carry is cleared.

Binary coded decimal

Two special rotate instructions are designed to deal with binary coded decimal, or BCD values. Binary coded decimal is a convention that is used to store two values, each in the range 0-9, in one byte. Each decimal value occupies either the upper or lower four bits of a byte. To assist the handling of such numbers, there are instructions for rotating a byte four bits at a time:
RRD (Rotate Right Decimal): a byte which is pointed to by HL is the target byte of this instruction. Its lower four bits are moved into the lower four bits of A. The upper four bits of byte (HL) are moved to its lower half, and the original contents of the bottom half of A replace them.
RLD (Rotate Left Decimal): the upper bits of (HL) are moved to A. The lower bits move up to take their place, and they are replaced with the old low bits of A.
The flags are affected as with a normal rotate, but carry is unaffected.
Although it is not a rotate instruction, it is worth mentioning an instruction that facilitates BCD addition and subtraction:
DAA (Decimal Accumulator Adjust): if used immediately after an addition or subtraction, DAA alters the contents of the A register to convert it to a BCD result. If A had contained 55 hex and the instruction ADD A,16 hex had been performed, the normal result, 6B, would be modified by DAA to become 71 hex. This instruction uses the state of carry and auxiliary carry to arrive at the correct answer. All flags are affected.
The Z80 is equipped with a terminal called the I/O pin. It can be used in conjunction with the address bus and external hardware to provide *input* and *output* ports. These are read from and written to in the same manner as memory locations. They offer a separate I/O map for the addition of

hardware devices such as video display chips, printer ports and the like. The advantage of using I/O ports for hardware is that they do not use up space in the memory map, and the decoding circuitry is actually made simpler by the use of the I/O pin. To send and receive data via the I/O map, certain instructions are provided:

IN A,port: the port number is a byte of data following in program memory. This is placed on the low eight bits of the address bus, the I/O pin asserted, and a read signal sent out. The data which is then presented on the data bus is loaded into the accumulator.

IN r,(C): C contains the port number for this instruction, which loads data from a port into the specified general purpose register.

OUT A,port: the method of sending data to a port. The port number is provided as immediate data in program memory.

OUT r,(C): allows data in general purpose registers to be sent to port number (C).

An interesting feature of the Z80 is brought to light by the Amstrad. When an OUT (C) or IN (C) instruction is executed, the value of the C register is placed on the low eight address lines for decoding as the port number. At the same time, the value held in B is placed on the high eight address lines. The Amstrad hardware actually decodes the upper half of the address bus to find its port numbers, so it is B that must be loaded with the number of the port that you wish to address — the value in C is ignored. We must still call the instructions by the names OUT (C) and IN (C) unfortunately; these are the standard Zilog mnemonics which are observed by assembler and monitor programs.

More complex instructions

Certain, rather specialised, tasks have been speeded up on the Z80 by the inclusion in the design of some powerful, if complex, instructions. The most commonly used of these are the block move instructions, intended to assist the transfer of data from one area of memory to another:

LDI (*Load and Increment*): load the memory location of which the address is (DE) with a byte of data of which the address is (HL). After the operation, the memory pointers DE and HL are incremented; BC, which is intended for use as a counter, is decremented.

LDD (*Load and Decrement*): load the memory location of which the address is (DE) with a byte of data from address (HL). After this operation, decrement the registers BC, DE and HL.

LDIR (*Load, Increment and Repeat*): an LDI is performed, and if BC has not become zero, the operation is repeated until it does. The effect of this instruction is to load a block of bytes, the length of which is contained in BC

from an area of memory beginning at (HL) into another area beginning at (DE).

LDDR (*Load, Decrement and Repeat*): similar to LDIR, but, because the memory pointers are decremented, they begin by pointing to the last byte to be moved and the operation works 'top down'. Both repeating codes are available so that it is possible to transfer data between blocks of memory that overlap.

Most flags are not affected by the LD block instructions; auxiliary carry is always cleared, and the P/O flag is set if BC has not been decremented to zero.

Other instructions in this category are:

CPI (*Compare and Increment*): A holds a value which is compared with (HL), setting the Z flag accordingly. HL is then incremented. BC, used as a counter, is decremented.

CPD (*Compare and Decrement*): as CPI but after the comparison, HL is decremented as well as BC.

CPIR: CPI followed by a repeat if BC is not zero and a match between (HL) and A has not been detected by the Z flag.

CPDR: decrement version of CPIR.

The carry flag is unaffected, but S, Z, and AC are all altered by the comparison, and the P/O flag is set if BC reaches zero.

INI: input the byte from I/O port (C), and store it at address (HL). Increment HL and decrement B only, thus leaving C unchanged.

IND: INI with HL being decremented.

INIR: repeating version of INI; it continues until B reaches zero.

INDR: repeating version of IND.

OUTI: as INI, but the byte from (HL) is written to port (C).

OUTD: decrement version of OUTI.

OTIR, OTDR: repeating versions.

The I/O instructions leave most of the flags undefined; carry is unaffected, and Z set according to the value read or written.

Certain memory locations at low addresses have a special significance to the Z80 microprocessor, because a group of instructions named RST (restart) make them very useful. RSTs are only one byte in length but replace the action of a three-byte CALL instruction. Software writers can place the beginning of often-used routines in these special addresses and therefore save space in their programs. There are eight RST instructions and on the Amstrad RST 30 can patched for your own purposes:

RST 0, 8, 10, 18, 20, 28, 30, 38 — the values indicate the hex address that is to be CALLed.

Interrupts

The Z80 CPU has an interrupt system which allows hardware signals to halt the processing of the current program and divert attention to another routine. It is intended to allow the servicing at regular intervals of external devices that need to be looked at, such as the keyboard. Two forms are implemented: *maskable* interrupts can be treated in a number of different ways and software instructions can disable them. *Non-maskable* interrupts work via a separate pin, and, as their name implies, cannot be ignored. NMIs cause the current value of the PC to be saved on the stack; they disable all other interrupts and force a jump to the memory location 66 hex. There is one instruction which works with non-maskable interrupts:

RETN: returns from a NMI subroutine, and sets the interrupt hardware to allow further interrupts.

Maskable interrupts, triggered by a signal to the CPU's INT pin, can be handled in one of three ways, determined by the use of a software instruction:

IM 0: requires a code to be sent back to the CPU on the data bus; the code is expected to be one of the RST instructions. Mode 0 is not used by the Amstrad.
IM 1: when an interrupt is received, an RST 38 is performed: this is the mode in which the Amstrad runs.
IM 2: the PC is pushed to the stack, the low eight bits present on the data bus are combined with the contents of the I register. This address is then used as a pointer to fetch an address to which control is then passed. Mode 2 is not used by the Amstrad.
Three commands are associated with maskable interrupts:
RETI: returns from a maskable interrupt routine.
DI: causes maskable interrupts to be ignored, or *disabled*.
EI: re-enables interrupts.

Two final commands

NOP: this code (00) does nothing; it simply occupies space in program memory. It is surprisingly useful when de-bugging programs.
HALT: when this code is encountered, the CPU stops processing instructions. It requires a RESET or interrupt to get the processor going.

The above descriptions cover all the instructions available on the Z80 processor. They are only building blocks for more complex operations: you will come across some of them time and time again; others are so rarely used that they are unfamiliar even to the most experienced programmer.

CHAPTER 2
Writing Code

Writing machine code programs involves assembling the building blocks of the available instructions into a program. Carrying out any major task will require a great many instructions, although there are ways in which we can cut down on the number needed.

Subroutines

Subroutines will be familiar to BASIC programmers. Z80 machine code is equipped with CALL and RET instructions: these allow us to organize a section of code which performs a particular task, and then CALL it as a subroutine from any part of the main program, or indeed from within another subroutine. This has two main advantages: we do not need to duplicate a sequence of instructions to achieve the same result, thus saving the number of bytes needed to store the program, and secondly, subroutines simplify the structure of a program, making it easier to write, understand and correct. One disadvantage is that you need to know exactly where in memory the subroutine is stored. The actual address must be included in the operation code for CALL, which is CD hex followed by the low and high bytes of the subroutine's address.

Loops

The use of reiteration is an important part of high-level computer languages. The BASIC FOR . . . NEXT loop command, the WHILE . . . WEND of Locomotive BASIC and other structures allows us to repeat sections of program, with altered variables, until certain conditions are met. In machine code, the conditional JUMP instruction, combined with operations to test a variable (or more than one variable) can be used to continue or terminate a section of machine code that performs a loop. Unlike subroutines, position-independent loops can be written, which means that code for one position will work if moved to another area of memory. To achieve this, the JR instruction, its conditional variations, and the DJNZ operation must be used. These limit the flags that can be tested, and restrict the size of the loop, although neither of these limitations is insurmountable.

Planning ahead

The first step in writing a machine code routine is the drawing of a flow diagram. Even if you have managed to write programs in BASIC without the aid of a flow diagram, please don't be tempted to do without them for anything other than the simplest of machine code tasks. The initial diagram need not be complete, tidy or well laid out; it need not be comprehensible to anyone but yourself; but you will find that some form of pre-planning is essential if you are to write efficient machine code. If the program is complex, begin by writing in plain English, and don't worry about the nuts and bolts of the program until you have an overall picture of what you wish to do.

Then you begin to sketch out how you might handle the trickier sections. Flow diagrams of these will help you to decide which registers to use, what subroutines are necessary, and what kind of data storage you will need to set up in RAM. This is also the best stage for considering alternative methods of solving problems; a quick look through the instruction set could remind you that, for example, you may be able to use the CPI command for a searching operation. If your task involves the use of many variable values, you should consider where you are going to keep them — it may be possible to keep them in registers and on the stack. If not, is it worth setting up an area of memory and pointing an index register to your new variables area?

Now the more pedestrian sections of the routine can be fitted around the special requirements with an eye towards the subroutines and data areas for which you have already determined a need. I find it difficult to resist the temptation to start coding the program at this point. If you do have patience, a tidy, properly laid-out version of your flow diagram will be a good investment in the long term, particularly if you annotate it with information such as register use, stack contents and the like. Don't be disappointed if your near drawing ends up having to be heavily modified, though. Trial and error, and the correcting of mistakes, are major parts of writing machine code.

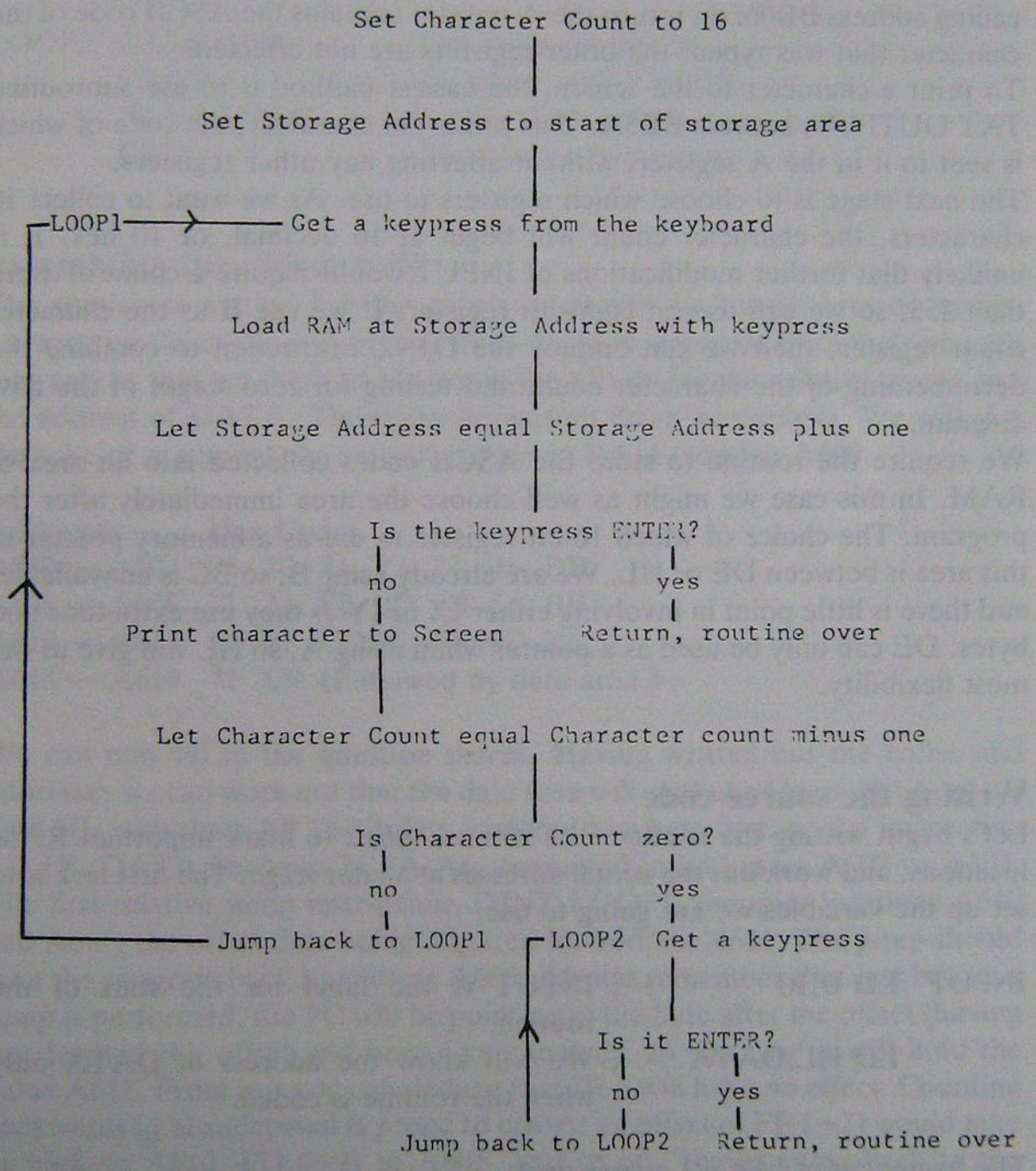
When your routine is finally entered, debugged and running, have another look. There is almost always room for some improvement in a machine code routine, even if it is just the trimming of a couple of bytes or a few machine cycles. In general, the less bytes a routine occupies, the faster it will run, unless you over-use loops and subroutines.

An example routine — INPUT

In order to demonstrate some of the principles involved, let's look at a simple example routine, which could have practical use as part of a larger program. Called INPUT, its task is to collect up to 16 characters typed in from the keyboard and place them in an area of memory. The letters that are input by the user are also printed to the screen, and hitting the ENTER key will cause

the routine to finish. Let's rough out a flow diagram (*see Figure 2.1*). Now to set about turning the theory into code. The two things that are going to be tricky are fetching characters from the keyboard and printing to the screen. Although it is quite possible, we do not actually need to write these ourselves — the Amstrad already contains subroutines in its firmware ROM which we can exploit. Both these tasks can be dealt with as subroutine CALLS.

Figure 2.1



There is a certain amount of jargon used when describing the use of subroutines. You can 'send' data to the subroutine in the registers. For example, 'A sent holding key number' means that you must load A with a value for the subroutine's information before calling it. Information is 'returned' from a subroutine in the same manner: 'result returned in HL' means that the subroutine has placed the answer in the HL pair. The flags are often used to pass information: 'carry returned false if the key is not pressed' is a good example of this. If a register is 'corrupted' it will not necessarily return with the same value it held when the subroutine was called. The routine for fetching a keypress, named KM WAIT CHAR, is reached by calling address BB06; on return the A register contains the ASCII code of the character that was typed: the other registers are not affected.
To print a character to the screen, the easiest method is to use subroutine TXT OUTPUT, address BB5A. This prints the character, the code of which is sent to it in the A register, without affecting any other registers.
The next stage is to choose which registers to use. As we want to collect 16 characters, the character count will begin at 16 decimal, or 10 hex. It is unlikely that further modifications of INPUT would require a count of more than 255, so we can use an eight-bit register. If we use B as the character count register, then we can employ the DJNZ instruction to combine the decrementing of the character count and testing for zero stages of the flow diagram.
We require the routine to store the ASCII codes collected into an area of RAM. In this case we might as well choose the area immediately after the program. The choice of which 16-bit register to use as a memory pointer to this area is between DE or HL. We are already using B, so BC is unavailable; and there is little point in involving either IX or IY as they use extra time and bytes. DE can only be used as a pointer when using A, so HL will give us the most flexibility.

Writing the source code

Let's begin writing the routine. We will use *labels* to mark important RAM locations, and work out the actual adresses at a later stage. The first task is to set up the variables we are going to use:

```
INPUT   LD B,10          ; INPUT is the label for the start of the
                           routine.
        LD HL,DATA       ; We will know the address of DATA only
                           when the routine is coded.
```

The beginning of LOOP1 comes next:

```
LOOP1: CALL BB06      ; Get a keypress into A.
       LD (HL),A      ; Store ASCII code in RAM.
       INC HL         ; Point HL to next location.
       CP 0D          ; Compare A with ASCII code for ENTER.
       RET Z          ; If they match, return from routine.
       CALL BB5A      ; Print character.
       DJNZ LOOP1     ; Decrement B; if it does not become zero
                        jump back to LOOP1.
```

The instructions for the second loop:

```
LOOP2: CALL BB06      ; Get keypress into A.
       CP 0D          ; Compare with ENTER.
       JR NZ,LOOP2    ; If they don't match, jump back.
       RET            ; The routine is complete.
```

Assembling the object code

The routine is writen, but the words must now be turned into op codes. Looking up the codes and writing them down in sequence is a simple enough job, except that we don't as yet know the offsets for the relative jumps and the address of DATA. These can be written down as symbols. We will, for the sake of argument, begin our routine at RAM location A600.

Addresses	Hex Codes
A600 — A607	06 10 21 ?? ?? 06 BB
A608 — A60F	77 23 FE 0D C8 5A BB
A610 — A617	10 ?? CD 06 FE 0D 20
A618 — A619	?? C9 (Followed by data area.)

We can now fill in the question marks. Having written out the codes and addresses we can work out that the data area will start at address A61A, so we load HL with that. All 16-bit data is stored low byte first, so the instruction LD HL,DATA becomes 21 1A A6, contained in addresses A602 to A604. The first relative jump instruction, DJNZ LOOP1, occupies locations A610 and A611, the offset data being required for address A611. The jump should take the program back to address A605. Always remember that just before a jump is performed, the PC will be pointing to the byte after the offset (having just fetched the offset and been incremented). In this case it will hold the value A612. From this you can deduce that JR 0 will have no effect. Counting backwards in hexadecimal is prone to errors: an offset of FE (−2) would take us back to A610, FD (−3) to A60F, and so on. Try counting back to the

location we wish to reach (A605): the answer for the offset should be −13 or F3. To calculate this value, subtract the PC from the target address; if you use the computer, the HEX$ function and & prefix will make this task easy. The second relative jump, in addresses A617 and A618 (JR NZ,LOOP2) can be worked out in the same maner. Have a go at this — the answer you should arrive at is given in Program 2.1.

Loading the routine

How do we get the bytes of machine code into memory? There are a number of methods, the simplest and most tedious of which is to set up a loop in BASIC that INPUTs the bytes and then POKEs them into place. *Program 2.1* is a little more sophisticated; the machine code instructions are held in DATA statements, thereby allowing them to be corrected if wrong. We will improve this method at a later stage to check for errors, but for now, the program is a quick way to test our short routine. Enter the listing and save it on tape. Now RUN it — after a short delay the machine code will come into operation.

Program 2.1

```
100 REM ******************************
101 REM **        INPUT Demo          **
102 REM ******************************
110 MEMORY &A5FF
120 RESTORE
130 FOR x=&A600 TO &A619
140  READ B$:POKE x,VAL("&"+b$)
150 NEXT x
160 CLS: PRINT "Press a key to run code"
170 IF INKEY$="" THEN 170
180 CLS: CALL &A600
190 PRINT "Code has finished"
200 STOP
210 REM ******************************
211 REM **        INPUT Data          **
213 REM **    (CHECK CAREFULLY!)      **
214 REM ******************************
220 DATA 06,10,21,1A,AB,CD,06,BB
230 DATA 77,23,FE,0D,C8,CD,5A,BB
240 DATA 10,F3,CD,06,BB,FE,0D,20
250 DATA F9,C9
```

Testing INPUT

INPUT behaves almost as if you were still in BASIC, except that the ESC key will do nothing. You might like to PEEK the data area to check that the characters have been stored. Notice that there is an inconsistency: even if you press ENTER the next printing occurs immediately after the 16th character. Here is a task for you — modify INPUT so that the print position is always moved to the start of the next line. This should be simple enough, but there is one possible pitfall that you may encounter (no clues!). You could also write a better version of INPUT in which the DEL key will function. You will find a list of the control codes and their functions in Chapter 9 of your Amstrad manual. The ASCII value of DEL is 7F.

The manual assembly of machine code is not only a tedious affair, it is very prone to errors. A good assembler program allows you to type in instructions using their mnemonics, the assembler names. You can edit, move and delete lines as if the instructions were a BASIC program, use special instructions to give values to labels, determine where the code will go, and even read in mnemonics from tape. When you are happy with the *source code*, as the collection of mnemonics is called, the program can turn it into machine code very quickly, the result being the *object code*. This avoids the possibility of human error and saves a lot of time, not to mention paper. If you enter a routine that will not work, the assembler program will, in most cases, blindly follow your instructions, but at least all those relative jumps will be right!

A monitor program allows you to examine and alter the contents of RAM. It is possible to get monitors that can also *disassemble* object code back into mnemonics and allow you to test routines one step at a time, showing all the register contents, the state of the flags and the important areas of memory after each instruction. With such a program, tracking down mistakes is made as easy as it can be.

This book was written with the aid of the Amsoft programs GENA3 and MONA3, the components of the *Hisoft Devpac* system. Although complex to learn to use, and not cheap, Devpac is an excellent, professional piece of software. *The Code Machine* from Picturesque is recommended for being easy to use, while in some respects it is even more powerful than *Devpac*. By all means consider other packages — but remember that the more powerful the program, the more useful it will be in the long run.

Despite what I have said, you won't need an assembler to use and understand the routines in this book. However, when you decide to create your own programs, let the computer do the tedious work for you.

Two things to avoid when writing machine code

One habit of inexperienced machine code programmers is to select areas of

RAM for use as variables that are within the program area, perhaps sandwiched between two subroutines. Try to keep all your variables in one, or possibly two, blocks before or after the program, rather than scattered about the routine. This tidy housekeeping will save many headaches when debugging your programs, and if you decide to use indexed addressing, you will be able to access all your variables easily.

Never be tempted to write self-modifying machine code. This form of program alters instructions within itself to cause different events to occur. Finding a fault in such a program can be extremely difficult, and you can never be sure quite what state the routine is in. There are some occasions when programmers can justify the use of self-modifying code — (relocating routines, program protection or lack of RAM), but never use the method unless you have to.

Both the above 'don'ts' will be brought home to you if you ever have a program that a software house decides to publish — on a ROM cartridge! Even if this seems unlikely to happen, it is good programing practice to consider your program area as if it were ROM.

CHAPTER 3
The Hardware and Firmware

Understanding the environment in which the CPU works is as important as understanding machine code itself. In order for the CPU to function, for us to be able to communicate with it, and for it to be able to respond in some meaningful way, many additional components are required.

Memory devices

The most important part of the computer, after the CPU, is the memory in which programs and data are stored. There are two types of memory used. Random access memory (RAM) can have its contents changed by the CPU, but 'forgets' the information it holds when the power supply is turned off. The Amstrad is equipped with 64K of RAM. Addresses 0000 to BFFF are used for general storage purposes. The 16K of RAM between C000 and FFFF is normally used as the video display memory area; pixel information is stored here, and used by other circuits to generate the video picture sent to the monitor.

Read only memory (ROM) cannot have its contents altered, but retains its data in the absence of a power supply. Therefore, it is used to store the built-in routines that run the computer. One area of ROM has addresses in the range 0000 to 3FFF; this is the firmware ROM, containing routines for handling screen, keyboard, sound, cassette and general house-keeping duties. A second ROM uses addresses C000 to FFFF. This is where the Locomotive BASIC interpreter resides: it can be considered as one, complex, machine code program that allows us to write and use BASIC programs. The BASIC machine code itself makes much use of the Firmware ROM. Models of the Amstrad that had disk drives fitted are equipped with further ROM memory.

The gate array

We have encountered a conflict: you can see from the above description that RAM and ROM share the same addresses. This means that circuitry is required to decide which type of memory should respond in a particular circumstance. Write operations are no problem — there is no point in trying

to send information to one of the ROMs — but hardware is required to enable the CPU to select its source for a read operation. This is contained in the *gate array*, an integrated circuit built especially for Amstrad.

Video display

The gate array chip handles a number of other functions, most of which are concerned with generating the video signals, which it does in conjunction with the cathode ray tube controller chip (CRT), a 6845 device. Between these two chips, which can both be controlled by the I/O operations of the CPU, the video display RAM is read and converted to a suitable analogue video signal. There are certain time constraints involved in this process, and in order for the gate array to access the video RAM directly, the machine cycle of the Z80 are manipulated so that a clash does not occur. The effect is completely transparent to the user — all that happens is that the CPU is slowed down slightly, running at an effective clock rate of 3.3Mhz, rather than 4Mhz.

Input and output

Other input and output circuitry, such as the cassette recorder interface, printer port, keyboard and sound chip, are buffered from the CPU by a general purpose input/output chip, the 8255 Parallel Peripheral Interface (PPI). This boasts a number of *ports*, not to be confused with the simple I/O port system of the CPU. Because these ports are separate from the computer's data bus we can make them 'latching', that is to say, set a bit of a particular port high and leave it in that state. For example, the signal that switches on the cassette motor stays high until a further signal is sent to the port to reset it. Port A sends and receives data from the sound chip; port B is a read only channel which detects signals from the cassette, printer port, and video circuits; while port C sends information to the cassette, control signals to the sound chip, and scanning signals to the keyboard. On machines fitted with a disk drive, a separate controller chip is provided.

Sound generation

The Programmable Sound Generator (PSG), a chip called the AY-3-8912, is a sophisticated device that can generate a wide range of sounds. Once sent the right information it will continue to produce sounds without further attention from the CPU. It also boasts a couple of I/O ports of its own, which are used to help implement the keyboard arrangement.

The keyboard

The keys and joystick connections are arranged as ten rows of eight keys. In order to read a row, a four-bit selection signal is sent from port C to the keyboard, which is decoded in order to select one of the ten rows. A byte of data, fed from the PSG I/O port, via the PPI, can then be read into the CPU. Each bit of the byte corresponds to a particular key in the row, and a decoding process is able to determine which key is pressed. The matrix arrangement is not too good at handling multiple key presses because the rows can interact, but the system is a cost effective way of producing a keyboard.

The Z80 can control all these devices through its I/O ports in combination with control signals in the same manner as it addresses memory locations. Using the devices is not a simple affair, but fortunately for us, the routines in the lower ROM make the task much less formidable.

Firmware

The Amstrad is an excellent machine on which to write machine code routines because not only are the routines that drive the hardware available, but also they are fully documented. The locations of the routines, which are published in the Firmware Specification, can be relied upon. Calls to the operating system pass through a table of JP instructions which are loaded into RAM when the system initialises itself. This table is called a *jumpblock*, while the operation is known as *vectoring*. By implementing this concept, Amstrad can re-write large chunks of the operating system without altering the firmware addresses. The practical outcome of this is that we can use the available ROM routines, and need not use low-level hardware driving routines of our own invention.

There is another reason for vectoring the firmware routines through a jumpblock — it is located in an area of RAM that does not share addresses with a ROM. Whenever a firmware routine is called, either from the BASIC ROM, the firmware ROM itself, or a machine code routine in RAM, which perhaps shares addresses with one of the ROMs, the process is the same. The RST addresses (see Chapter 1) hold the same information in RAM and ROM, and one of the RSTs (8), is coded so as to allow the ROM to be enabled. Before returning from the RST, the state of the ROMs is restored to what it was before the RST occurred. By placing the jumpblock in an area of RAM where it can always be reached, and providing software in the RST addresses which allows us to save and later to restore the current ROM enabling state, the routines can be called from anywhere in the memory map.

When a CALL is made from BASIC to a machine code routine, it is safe to assume that both ROMs are disabled. You can leave them in that condition,

even if you use the firmware. When you finally return to BASIC, you may leave all the normal registers in whatever state you wish, with exception, of course, of the stack pointer. As for the alternative register set, I mentioned earlier that it is best left alone because the firmware routines make use of it. Even if you do not use the firmware, the computer normally has interrupts enabled in order to keep its timers up to date, and these interrupts *will* use the firmware. If you really do want to use the alternative registers, it will be necessary to disable the interrupts, or alter the interrupt routines themselves. Although the people behind Locomotive Software are more secretive about their BASIC than their operating system, machine code can be used to add extended commands. How this can be done is demonstrated in Chapter 9.
We have used two firmware routines already, and many more will be put to use in later chapters. Wherever I introduce a firmware routine, as well as the address I will give the name and details of which registers it corrupts. Many other useful routines will be explained, but it is beyond the scope of this book to list every single routine, particularly those with very specialised functions. Important calls to the firmware that are not listed in detail elsewhere in the book are given below, divided, as is the jumpblock, into a number of sections.

Key Manager

BB00	INITIALISE	Sets up keyboard. No entry conditions. Corrupts AF, BC, DE, HL.
BB06	WAIT CHAR	Waits for a character from the keyboard or an expansion string. (Expansion strings are part of the firmware provisions of the operating system; the DEF KEY command uses them to translate key tokens into strings.) No entry conditions. Character returned in A. Other registers preserved.
BB09	READ CHAR	As WAIT CHAR, but does not wait. If no character is available, carry is returned false.
BB18	WAIT KEY	As WAIT CHAR, but expansion tokens are returned rather than a character from the relevant expansion string.
BB1B	READ KEY	Returns a keypress without waiting: as with WAIT KEY, expansion tokens are returned.
BB1E	TEST KEY	Sees is a particular key is pressed. The key number is passed in A (see the manual for numbers). On return, if the specified key is pressed, Z will be clear: if CTRL is pressed, bit 7 of C will be set: and if SHIFT is pressed, bit 5

		will be set. Also corrupts A, HL.
BB21	GET STATE	Fetches the caps and shift lock states. L returns the shift and H the caps states, with 00 being off, FF on. Corrupts AF.
BB24	GET JOY	Returns a value relating to both joysticks. No entry conditions. H and A hold the state of joystick 0, and L the state of joystick 1, where bits 0 to 5 relate to up, down, left, right, F2 and F1 respectively. If the button is pressed, the bit is set. Other registers are preserved.

Further Key Manager calls are listed in Chapter 5.

Text VDU

BB4E	INITIALISE	Sets up the character screen. No entry conditions. Corrupts AF, BC, DE, HL.
BB5A	OUTPUT	Sends the character or control code in A to the screen. A study of Chapter 9 of the manual will show that control codes allow you to set text colours, move the print position, and much more. Preserves all registers.
BB5D	WR CHAR	Prints a character as *output*, but control codes are not obeyed: a series of symbols is printed instead.
BB60	RD CHAR	Reads the character at the current cursor position on the screen. Returns carry false if the character is unrecognisable, otherwise carry true and the ASCII code of the character in A. Preserves other registers.
BBB4	STR SELECT	Changes the stream number on which the TXT calls operate. The new stream number is sent to the routine in A, and the previous stream number is returned from the routine in A. HL is corrupted.

The Graphics section of the firmware is detailed in Chapter 7.

Screen routines

BBFF	INITIALISE	Sets up all the screen variables to their default values. Corrupts AF, BC, DE, HL.

BC0E	SET MODE	Changes the screen mode to the value sent in A. Corrupts AF, BC, DE, HL.
BC4D	HW ROLL	Scrolls the entire screen up or down one character line. If B is sent to the routine holding zero, the scroll will be downwards: A must hold the encoded ink for the new line. Corrupts AF, BC, DE, HL.
BC50	SW ROLL	Scrolls any specified area of the screen by copying. B holds the direction (0=down), A the encoded ink for the new line, while the area is defined as physical character co-ords — right, bottom, left and top in D, E, H and L respectively. Corrupts AF, BC, DE, HL.

Other screen routines are discussed in relation to the FILL routine in Chapter 8.

Cassette Manager routines

BB65	INITIALISE	Sets up the Cassette manager: corrupts AF, BC, DE, HL.
BC6B	NOISY	Controls the generation of cassette prompts. If A is sent as zero, the prompts are enabled, otherwise they are suppressed. Other registers preserved.

To write a binary file to cassette, the following routines should be used:

BC8C	OUT OPEN	Opens the file and sets up the header. B should contain the length of the filename, HL its address. If O.K., returns carry=1, zero=0.
BC98	OUT DIRECT	Sends data to the cassette. HL should hold the first address, DE the length, BC the 'call' address that a RUN command would enter a machine code routine at, and A the file type (2 for binary). If successful, returns carry=1, zero=0.
BC8F	OUT CLOSE	Close the file, ensuring the whole block has been sent. Returns carry=1, zero=0 if O.K.

To read the file back in, a 2K buffer needs to be reserved. The following routines should be used:

BC77	IN OPEN	Opens the file and fetches the first 2K into the buffer. B and HL should be set up for a

		filename as OUT OPEN, unless B=0, in which case the first file is read. DE must point to the buffer area. If carry=1, zero=0 returned, the file was successfully opened. Other information returned: HL=header address, DE=data location, BC=length and A=file type; all are taken from tape.
BC83	IN DIRECT	Loads the file. HL should contain the address to place the file. If O.K., carry=1, zero=0.
BC7A	IN CLOSE	Closes the file. Returns carry true if O.K.

Note that if you have a disk drive, the cassette calls will normally be overwritten with routines that use the disk drive — all the above should work with disks.

Sound

BCA7	RESET	Clears all queues and silences the sound chip. Corrupts AF, BC, DE and HL.

Using the sound manager is a complex affair, and of specialised interest. The simplest way to make a beep is to print a CHR$(7), the BELL control code, with TXT OUTPUT (BB5A).

Kernel Routines

These are mainly concerned with interrupts and *events*. Two routines which may be of general use are listed below.

BD0D	TIME PLEASE	Returns the current value of the timer in DEHL, a four-byte count in units of 1/300th of a second. Preserves other registers. Note that the timer is interrupt-driven: disabling the interrupts will stop the count.
BD10	TIME SET	Sets the timer count to the value held in DEHL. Corrupts AF.

The Machine pack

BD19	WAIT FLYBACK	Waits for a video flyback pulse. Useful for synchronising graphics with the video signal thereby avoiding flickering effects.
BD2B	PRINT CHAR	Sends the value in A to the Centronics printer

		port. Bit 7 of A is ignored. On return, if carry=1 then all went OK. If carry=0 then the routine 'timed out': i.e. the printer was busy for over 0.4 secs. A is corrupted.
BD34	SOUND REG	Sets a register of the AY-38912 PSG. A should hold the register number, C the data. Corrupts AF, BC. Useful for talking directly to the PSG.

If you intend writing a routine that will reside in ROM, or wish to make use of interrupts, you will need to study the full firmware specification with care. For our purposes, however, the above information, plus that given in later chapters, allows you to use most of the Amstrad's hardware with confidence.

CHAPTER 4

Machine Code Number Sorting

The main advantages of keeping data in a computer are the speed with which a specific item can be found, and the ability to sort items into a particular order. The stock control of a warehouse, or the membership records of a large organisation are examples of applications that spring to mind, but more humble tasks are equally suitable: keeping an address book that allows you to print envelopes for this year's Christmas cards, or cataloguing collections alphabetically, chronologically or even by value.

For general applications you could purchase a database program that will fulfill your needs, but writing your own program gives you the opportunity to match it to precise requirements, and also allows you to expand the program as you become aware of special circumstances. BASIC is a perfectly suitable language in which to write the main body of your program. Inputting information, saving and loading from tape and displaying records on the screen can all be handled quickly enough, but BASIC is not nearly as fast as machine code when it comes to sorting data. This chapter presents a number of routines that will sort BASIC variable arrays into order with an astonishing saving of time. The techniques can easily be applied to data that is not held by BASIC, so writers of pure assembly language will also find them of use.

Sorting methods

Before presenting the first routine, I think a look at how we would sort an array using BASIC will be of benefit. The method is the same as we will use for our first machine code sort. The simplest way to sort data is to use a method, or algorithm, called a *bubble sort*. This rather picturesque name is meant to conjure up a vision of higher valued items rising to the top of an array as we process it.

The human approach

If you were to give a person a hand of playing cards, they might put them into order by searching through for the card of lowest value. When they had ascertained which of the cards this was, they would remove it from its current

position and place it at the end of the hand. They would then scan the remaining cards to discover which one is now the lowest and, when found, that card would be placed alongside the card first removed. Each time the search of the remaining cards is successfully completed, the number of sorted cards grows and the residue gets smaller. Finally, the hand will be in the correct order.

Of course, human intuition would play a large part in the sorting of a hand of cards. For example, by knowing that a card has the lowest value we could place it as soon as it is found or we might recognise that a group of cards is already in order.

The computer approach

To imitate most of the short cuts a human might take would involve some very complex programming. The bubble sort is a simple method that a computer can easily handle, while more advanced algorithms can exploit a computer's strengths and minimise the effect of its weakness.

Look at the Integer Sort Demo program, Program 4.1. Lines 1160 to 1230 make up a BASIC bubble sort routine. From this you can see the technique that we will also employ in the first machine code sorting routine.

Program 4.1

```
1000 REM ********************************
1010 REM **    Integer Sort Demo.     **
1020 REM ********************************
1030 GOSUB 8000
1040 WHILE 1:SIZE=0
1050  WHILE SIZE<1
1060   CLS:INPUT "Size of array";TMP%
1070   SIZE=TMP%-1
1080  WEND
1090  DIM STORE%(SIZE),SORT%(SIZE)
1100  FOR VALUE=0 TO SIZE
1110   STORE%(VALUE)=32768-RND*65535
1120  NEXT VALUE
1130  GOSUB 2000
1140  PRINT "BASIC bubblesort first - please wait."
1150  START=TIME
1160  FOR PASS=SIZE TO 1 STEP-1
1170   FOR COMP= 0 TO PASS-1
1180    IF SORT%(COMP)>SORT%(COMP+1) THEN 1220
1190    TMP%=SORT%(COMP)
```

```
1200     SORT%(COMP)=SORT%(COMP+1)
1210     SORT%(COMP+1)=TMP%
1220    NEXT COMP
1230   NEXT PASS
1240   BASDUR=TIME-START
1250   PRINT "BASIC bubblesort took ",BASDUR/300;" seconds
1260   GOSUB 3000:GOSUB 2000
1270   PRINT "SortInt next - please wait"
1280   START=TIME
1290   CALL HIMEM+1,@SORT%(0),SIZE+1
1300   SIDUR=TIME-START
1310   GOSUB 3000
1320   PRINT "BASIC bubblesort took ",BASDUR/300;" seconds
1330   PRINT "SortInt took ",SIDUR/300;" seconds"
1340   PRINT "Another array?  (Y/N)
1350   TMP$=INKEY$:IF TMP$="" THEN 1350
1360   IF TMP$<>"Y" AND TMP$<>"y" THEN END
1370   ERASE STORE%,SORT%
1380 WEND
2000 REM ******************************
2010 REM **       Copy Array         **
2020 REM ******************************
2030 FOR VALUE=0 TO SIZE
2040  SORT%(VALUE)=STORE%(VALUE)
2050 NEXT VALUE
2060 RETURN
3000 REM ******************************
3010 REM **       Print Array        **
3020 REM ******************************
3030 PRINT "Do you want to see the results?  (Y/N)"
3040 TMP$=INKEY$:IF TMP$="" THEN 3040
3050 IF TMP$<>"Y" AND TMP$<>"y" THEN RETURN
3060 PRINT "Old array",,"New array"
3070 FOR VALUE=0 TO SIZE
3080  PRINT STORE%(VALUE),,SORT%(VALUE)
3090 NEXT VALUE
3100 RETURN
8000 REM ******************************
8010 REM **   Code Loading Routine   **
8020 REM ******************************
8030 RESTORE:READ TOT:TLY=1
8040 FOR CHK=HIMEM+1 TO HIMEM+TOT STEP 8
```

```
8050  SUM=0
8060  FOR BLK=0 TO 7
8070   READ B$:TMP=VAL("&"+B$)
8080   SUM=SUM+TMP
8090   IF TMP<>PEEK (CHK+BLK) THEN TLY=0
8100  NEXT BLK
8110  READ TMP:IF TMP=SUM THEN 8130
8120  PRINT "Data error in block ";(CHK-HIMEM)8+1:STOP
8130 NEXT CHK
8140 IF TLY THEN RETURN
8150 RESTORE:READ TOT:MEMORY HIMEM-TOT
8160 FOR ADR=HIMEM+1 TO HIMEM+TOT STEP 8
8170  FOR BLK=0 TO 7
8180  READ B$:POKE ADR+BLK,VAL("&"+B$)
8190  NEXT BLK:READ TMP
8200 NEXT ADR
8210 RETURN
9000 REM ********************************
9010 REM **        SortInt Data          **
9020 REM ********************************
9030 DATA 104
9040 DATA DD,4E,00,DD,46,01,50,59,760
9050 DATA DD,6E,02,DD,66,03,23,7E,820
9060 DATA C6,80,77,23,23,1B,7A,B3,843
9070 DATA 20,F5,0B,C5,DD,6E,02,DD,1039
9080 DATA 66,03,54,5D,13,13,1A,96,496
9090 DATA 13,23,1A,9E,38,10,C5,46,577
9100 DATA 1A,77,78,12,1B,2B,46,1A,449
9110 DATA 77,78,12,13,23,C1,23,13,558
9120 DATA 0B,79,B0,20,E1,C1,0B,79,890
9130 DATA B0,20,D0,DD,5E,00,DD,56,1038
9140 DATA 01,DD,6E,02,DD,66,03,23,695
9150 DATA 7E,D6,80,77,23,23,1B,7A,806
9160 DATA B3,20,F5,C9,00,00,00,00,657
```

The method comprises two loops. The outer loop begins by subjecting the whole array to the inner loop, on the assumption that nothing is in order. The inner loop begins at the start of the array and compares the first and second items: if the first is lower than the second then they are swapped over. This inner compare loop then proceeds to perform the same operation on the second and third items and continues until it has compared (and swapped if necessary) all the adjacent items in the unsorted section of the array, which

on the first pass is the whole array. At the end of this process we can guarantee that the last item in the section was the lowest value found: on the first pass of the inner loop, we have created a sorted section consisting of one item, the last one.

The second time that the outer loop subjects the array to the compare loop it does not include the last item, the third time it can ignore the last two items, and so on. For each pass the sorted section becomes one item larger, the unsorted section one item smaller. Eventually, the number of items in the unsorted section is reduced to one, which, as all the other items were of a lower value, must be the highest.

That is the principle behind a bubble sort, and when programmed in BASIC it seems quite neat and compact. Indeed, sorting a dozen or so items into order can be done quickly, but as we increase the number of items in the array things begin to slow down dramatically. For each extra item added to the array, not only is an extra inner, or compare loop required, but that loop is one comparison, and potential swap, longer. The mathematics of the situation are such that multiplying the size of an array by ten multiplies the time required to sort it by one hundred.

Integer array sorting

There is no better way of showing how bubble sorts slow down with increased array size than by a practical demonstration. Type in Program 4.1 and save it, as you will need to modify it to test further routines in this chapter. The program compares a sort programmed in BASIC with a machine code version, the listing of which is given in Source listing 4.1. Both methods sort an array of integer values into high to low order. Integer variables (defined with the % symbol) are whole numbers in the range −32768 to +32767, and therefore only two bytes of RAM are required to store each value. They are the simplest type of BASIC variable, and so a good place to start.

Source code listing 4.1

```
                100 ; SortInt
                110 ; Sorts a BASIC integer array into high to low order.
                120 ; Requires the address of the first element and the
                130 ; length of the array to be passed by BASIC.
                140 ; Position independent - corrupts registers AF,BC,DE,HL.
DD4E00          150 SORTI  LD   C,(IX+0) ;Get length of array into BC and DE.
DD4601          160        LD   B,(IX+1)
50              170        LD   D,B
59              180        LD   E,C
DD6E02          190        LD   L,(IX+2) ;Point HL to first item in array.
DD6603          200        LD   H,(IX+3)
23              210        INC  HL       ;Point to MSB of value.
7E              220 UNSGD  LD   A,(HL)   ;This loop converts the array from
```

```
C680    230        ADD  A,#80     ;signed to unsigned numbers.
77      240        LD   (HL),A
23      250        INC  HL
23      260        INC  HL
1B      270        DEC  DE
7A      280        LD   A,D
B3      290        OR   E
20F5    300        JR   NZ,UNSGD
0B      310        DEC  BC        ;Set BC to one less than array size.
C5      320 PASS   PUSH BC        ;Save outer loop count.
DD6E02  330        LD   L,(IX+2)  ;Point HL to first item.
DD6603  340        LD   H,(IX+3)
54      350        LD   D,H       ;Point DE to second item.
5D      360        LD   E,L
13      370        INC  DE
13      380        INC  DE
1A      390 NEXT   LD   A,(DE)    ;Compare adjacent items
96      400        SUB  (HL)
13      410        INC  DE
23      420        INC  HL
1A      430        LD   A,(DE)
9E      440        SBC  A,(HL)
3810    450        JR   C,DONE?   ;If no swap, jump to DONE?
C5      460        PUSH BC        ;We will use B as a temporary store.
46      470        LD   B,(HL)    ;Swap over MSBs.
1A      480        LD   A,(DE)
77      490        LD   (HL),A
78      500        LD   A,B
12      510        LD   (DE),A
1B      520        DEC  DE        ;Swap over LSBs.
2B      530        DEC  HL
46      540        LD   B,(HL)
1A      550        LD   A,(DE)
77      560        LD   (HL),A
78      570        LD   A,B
12      580        LD   (DE),A
13      590        INC  DE        ;Restore pointers.
23      600        INC  HL
C1      610        POP  BC        ;Recover .counter.
23      620 DONE?  INC  HL        ;Point to next items.
13      630        INC  DE
0B      640        DEC  BC        ;Count minus one.
79      650        LD   A,C       ;Check for zero.
B0      660        OR   B
20E1    670        JR   NZ,NEXT   ;If not end of count, repeat NEXT.
C1      680        POP  BC        ;Retrieve outer loop count.
0B      690        DEC  BC        ;Count minus one.
79      700        LD   A,C       ;Check for zero.
B0      710        OR   B
20D0    720        JR   NZ,PASS   ;If not end of count, jump to PASS.
DD5E00  730        LD   E,(IX+0)  ;This section returns the array to
DD5601  740        LD   D,(IX+1)  ;its original form.
DD6E02  750        LD   L,(IX+2)
```

```
DD6603      760         LD    H,(IX+3)
23          770         INC   HL
7E          780 SIGND   LD    A,(HL)
D680        790         SUB   #80
77          800         LD    (HL),A
23          810         INC   HL
23          820         INC   HL
1B          830         DEC   DE
7A          840         LD    A,D
B3          850         OR    E
20F5        860         JR    NZ,SIGND
C9          870         RET
```

The BASIC program

Before we run the program and then examine the machine code, have a look at the BASIC. Please take particular note of the last section, after line 8000 — it is the code loading routine that we will use for all our BASIC programs. At a later stage you will probably wish to use an assembler to create machine code, but for now the BASIC loader will suffice.

Program 4.1 begins by calling the code loading routine at line 8000, which installs the machine code above a lowered HIMEM location. The main program then asks for an array size and sets up two arrays. One, STORE%, is loaded with random numbers; the other, SORT%, is the array that is acted upon by both the BASIC and the machine code sorting routines. The contents of STORE% are copied into SORT% and a note made of the timer value. The BASIC bubble sort then takes place, and the time taken calculated and stored. The subroutine beginning at line 3000 is called, which gives you the chance to compare the contents of STORE% with the newly sorted SORT%. Next, the original state of SORT% is restored by copying STORE% into it (subroutine, line 2000), the timer value noted and the machine code sorting routine called. The duration of the sort is calculated; you are given the opportunity of studying the effect of SortInt and the relative times of the two methods are then displayed. Finally, you are asked if you wish to repeat the whole process.

Using the integer sort demonstration

When you have entered and safely saved the program, RUN it. There will be a pause while the machine code is checked, and if you have made an error when entering the DATA statements it will be at this point that you will normally be informed. I say normally because the method used to check the data is not completely foolproof. It is possible for two errors within the same block of data to cancel each other out, so if the machine code routine crashes

or does not work as described then this may be the cause. For this reason, as well as the possibility of a power cut, it is always wise to save programs as soon as they are entered.

If the code is O.K. then it will be loaded above HIMEM. The next stage is for the program to ask you how large an array you wish to sort; to begin with, try 50 items. The BASIC sort should take about 14 seconds: the machine code will do the same job in about 0.07 of a second — 200 times faster! Try a few arrays of larger sizes to see how the speed of the sort decreases dramatically. Up to a certain point the time saved by using machine code will be in the same proportion, but with very large arrays BASIC will slow down even more, as it will often have to perform the 'garbage collection' required to remove out-of-date variables each time that it runs out of variable space.

You use the routine with the BASIC CALL command. In the demo program the machine code starts at HIMEM+1, and you can see from line 1280 that two parameters are required: the first (@SORT% (0) in the program) passes the address of the first element of the array you wish to sort, the second (SIZE+1) gives the number of elements, counting from one (so if your array has, say, 6 items, 6 is passed as the second parameter).

The use of SortInt (the name I have given the routine) is not restricted to the above BASIC program. To include lines 8000 onwards in any other program and a GOSUB 8000 at the start of the program is the simplest way to install the routine, but it will be quicker if you save the code as a binary file and load it into memory from cassette or disk. You may load it anywhere sensible: the code is completely relocatable. If you are not sure exactly how to do this you will find an example in Chapter 6.

And so to the machine code itself. The routine begins, at assembler line 150, by fetching the length value passed by BASIC. Parameters that have been sent by the CALL command can be found in a reserved area of memory, the first byte of which is pointed to by the IX register; they are in the reverse order to that of the list following the CALL, so that, as the length is the last parameter passed by BASIC, it follows that we will find the 16-bit length of the array at addresses IX and IX+1. This is loaded into BC, which is to serve as the outer loop counter, and also into DE. HL is filled with the second parameter, the address of the first item of the array, and then incremented so that it points to the most significant byte (MSB) of that item.

Handling Signed numbers

Integer array values are stored in the same manner as Z80 16-bit numbers, that is, low byte first, but there is the complication of negative numbers. These are held as 16-bit two's complement values: &FFFF is −1, &FFFE is −2 and so on. If we were to compare the numbers as they stand, some means

of recognising the numbers were negative would be needed; unless we did so, 8000 (−32768 decimal) would appear to be higher than 7FFF (+32767 decimal). A quick method is to add 8000 hex to all the values before we try to sort them, as this gives all the values the correct relationship. 8000 becomes 0000 (using 16-bit arithmetic means we lose the overflow): 0001 becomes 8001; and 7FFF takes on the value FFFF. When we have sorted the numbers we can restore them to their original value by subtracting 8000 hex.

Lines 220 to 300 are the instructions that perform the conversion. DE is used as a counter: each value has &80 added to its MSB (there is no point in adding 00 to the LSB), the pointer in HL is incremented to point to the MSB of the next value, and then DE is decremented and tested to see if it has become zero. If it has not, the unsigned loop (i.e. the loop for converting the values to unsigned numbers) continues.

Note how we test DE, setting the Z flag if it is zero so that JR NZ can be used — it is a common mistake to forget that decrementing register pairs does not affect the flags. More often than not this works to the programmer's advantage, but in this case it means we must use LD A,D and then OR E to set the Z flag if DE has reached zero.

With the array converted to an easily comparable form, we enter the outer loop after decrementing BC so that it equals the length of the array minus one. This outer loop counter is saved on the stack, although its value, left in BC, will also be used as the initial value of the inner loop counter. HL is pointed to the low byte of the first item, DE to the second.

Now the inner loop begins. The byte at address (HL) is subtracted from that at (DE). The result is discarded, but the value of the carry flag (showing if a 'borrow 256' was generated) will be carried over to the next subtraction. The pointers are incremented to point to the MSBs of the two values; this shows two of the oddities of Z80 code in their true light. In contrast to when we wished to know if DE had been decremented to zero (line 300), here we are thankful that the flags are not altered by the INC register pair instructions. Secondly, if we are working 'low to high' through a block of two-byte numbers, the advantage of the way the Z80 stores 16-bit numbers is shown — if they were stored in the order MSB-LSB we would have to juggle with the pointer in HL so as to extract the low byte first.

The next stage (lines 430-440) is to SBC (subtract with carry) the byte at (HL) from that at (DE), thereby taking into account the posibility of a 'borrow 256' being generated by the LSB subtraction. After this instruction, the carry flag will represent a true comparison between the two values. If it is set, then the first value is greater than the second, so no swap is required and line 450 causes the next stage, the swapping of the values, to be skipped over.

Swapping the values

In order to swap over the two values we need some extra storage space. The quickest way to gain a register is to push the contents of BC (at this stage the inner loop counter) onto the stack and retrieve them later. Other methods that may be considered include using IY, but index register instructions are long-winded both in time and bytes.

The operation between lines 470 and 610 may seem a little cumbersome at first glance but the instructions are only one byte in length, and most only take 4 clock cycles to perform. First, the byte from (HL) is stored in B (LD B,(HL)). We can now fetch the byte from (DE) and put it at (HL) with the instructions LD A,(DE) and LD (HL),A. The old value that was at (HL), now safely stored in B, can be loaded into the address (DE) with LD A,B and LD (DE),A. (Note how the lack of a LD (DE),B instruction means that we have to transfer the value to A.) Having swapped over the MSBs, the pointers are decremented, the LSBs swapped in the same manner, and the pointers restored.

The routine reaches assembler line 620, label DONE?, having compared, and swapped if necessary, one pair of adjacent items. This is the end of the inner loop. HL and DE are pointed to the next pair of variables, and then we decrement the inner loop counter and test for zero. If zero is reached we have completed the inner loop, otherwise the routine jumps back to the NEXT label.

The outer loop

Remember that in line 320 we saved the outer loop counter on the stack. The inner loop has exhausted its counter. We POP BC from the stack, so B now contains the outer loop counter. The bubble sort demands that we repeat the inner loop one time less than there are items to be sorted, and it is this function that the outer loop controls. For example, if the array held 5 items, the inner loop would need to be repeated four times. On the first pass the inner loop needs to process all the items, so its counter needs to start at 4. When we retrieve the outer loop counter, it is decremented; if it has not reached zero, then another pass is required. If we now jump back to the instruction where the outer counter is saved on the stack, we can use the outer counter as the new starting value for the inner loop. The inner loop counter is therefore loaded with diminishing values for each pass, which it takes from the outer loop counter.

Ending the routine

When, eventually, zero is reached by the outer loop counter, all that remains

is to return the array to its original form by converting the array contents back to signed numbers. HL and DE are reloaded from the parameter area and lines 780-860 (the signed loop) reverse the process of the unsigned loop (lines 220-300). Finally the RET instruction takes us back to BASIC.

What do you do if you want to sort an array in the reverse order? No problem — to modify SortInt to work the other way round you only need to change one byte. Line 450 holds the key: JR C,DONE? (38 10). This is the condition test that bypasses the swap section of the routine. At this point in the routine the carry flag will be set if the first value is greater than the second, which will *not* cause a swap. By changing line 450 to JR NC,DONE? (30 10) then a swap *will* be carried out of the item is greater than the second, resulting in the whole sort working in the opposite direction.

The bubble sort is capable of a simple enhancement that can speed up the sorting of arrays which are already nearly sorted. If the inner loop were to process a whole pass without having to swap any items, this would be an indication that the array was in order. A flag can be set at the start of the inner loop, and reset if a swap occurs. If the outer loop detects that the flag has remained set, it knows that the array is sorted. You may like to add this check to the above routine — it will test your ingenuity as there is no obvious place in which to store the flag.

Sorting floating-point arrays

The next routine is designed to sort an array of real numbers — the name used to define numbers that have both integer and fractional parts — sometimes known as *floating point* numbers. In order to represent these, Locomotive BASIC uses five bytes. The format is capable of storing numbers in the range 2.9E-39 to 1.7E+38 to nine-digit accuracy.

Obviously, comparing two blocks of five bytes will take longer than comparing integer variables, and swaps will take longer too. It is time to refine our sorting technique a little; a small modification to the algorithm will reduce the number of swaps we need to perform by a considerable amount. This new method is called a *delayed replacement sort*. There are more advanced methods available, such as the Shell-Metzner algorithm, but these only come into their own with quite large arrays, and may even be slower for small ones. Programming them in BASIC is worthwhile. The extra complexity of the machine code versions is unlikely to pay dividends until array size approaches the capacity of your computer.

As a bubble sort progresses, there are a lot of swaps that are redundant. If one item is of a lower value than, say, the five items above it in the array, it will change places with each in turn. The delayed replacement sort avoids this by comparing the first item with subsequent values until it finds a value with

which a swap is *not* required. It then notes the address of this new value as the lowest it has found to date, and continues its search, updating the lowest value address each time it finds a lower value. Only when the end of the unsorted section of the array is reached are the values swapped over — the lowest value found is exchanged with the last unordered item, the size of the unordered area is reduced by one, and then this comparison loop is repeated until the whole array is sorted.

This method is much closer to my earlier description of the way cards can be ordered. As a basis for a machine code routine it has the drawback of being more complex to program, but it is nevertheless feasible, and the speed gained compensates for the extra work required to handle real numbers: sorting a real array using the delayed replacement method in machine code takes about twice as long as a SortInt spends on an integer array.

Real sort demonstration

Program 4.2 is the data required to load the machine code. We will use the BASIC from the integer program, 4.1. You must change the array names so that real arrays are used (that is, delete the % suffix used for integer numbers, and if you wish, add a ! suffix). To give a proper test, change the random number generator so that it gives a greater range.

Load Program 4.1 and make the required changes: delete lines 9000-9160 and replace them with Program 4.2.Save and then RUN the new program, SortReal Demo. If the data statements are O.K. then you will be able to test the new routine against a suitably modified bubble sort, or if you wish you can write a BASIC DR routine. It would be a good test of your understanding of the principle to do this — convert the machine code to BASIC!

The format of real numbers

Before we look at the machine code, I must describe how Locomotive BASIC uses the five bytes to represent floating point numbers. One byte represents the scale of the number and the others give the fractional part. To explain this let me give a simplified view first; the value 1117 can be written as 1.117 * (10 to the power of 3). One byte of the floating point representation would hold the (10 to the power of 3) scaling value (called the 'exponent'), while the other bytes would store the .117 (called the 'mantissa').

The order in which the bytes are stored are the four mantissa bytes, least significant first, followed by the exponent. The most significant of the mantissa bytes (the fourth) has its seventh bit used for indicating the sign — if set then the number is negative.

I said that the above description was simplified; if you look at the values

Program 4.2

```
9000 REM ******************************
9001 REM **        SortReal Data
9002 REM ******************************
9010 DATA 128
9020 DATA DD,4E,00,DD,46,01,0B,C5,799
9030 DATA 01,04,00,DD,6E,02,DD,66,661
9040 DATA 03,09,54,5D,09,EB,2B,C1,669
9050 DATA C5,C5,D5,E5,1A,AE,17,30,1107
9060 DATA 04,1A,17,18,2B,23,7E,A7,448
9070 DATA 28,F7,13,1A,A7,20,06,2B,580
9080 DATA 7E,17,3F,18,1B,1A,96,20,471
9090 DATA 12,01,FC,FF,EB,09,EB,09,1014
9100 DATA 06,04,1A,96,23,13,1A,9E,424
9110 DATA 10,FA,17,2B,46,1F,A8,17,624
9120 DATA E1,D1,30,02,62,6B,C1,0B,893
9130 DATA 78,B1,28,0A,C5,01,05,00,550
9140 DATA EB,09,EB,C1,18,B3,13,23,929
9150 DATA 06,05,1A,4E,77,79,12,2B,416
9160 DATA 1B,10,F7,C1,0B,78,B1,20,823
9170 DATA 8E,C9,00,00,00,00,00,00,343
```

actually stored, you will find that they are somewhat more complex. First, the exponent byte gives a scaling value to the base two, and in order to be able to express both large and small values takes a mid-point in its range (81 hex) to define an exponent of 1. Numbers with this exponent will therefore fall in the range 1 to just under 2. An exponent of 82 puts the value in the range 2 to just under 4; 83 gives 4 to just under 8 and so on. Going down the scale, 80 dictates a value between .5 and just under 1, 7F gives .25 to just under .5. The mantissa part of the number gives the fractional part within the range specified by the exponent. Thus a mantissa of 0000 hex and exponent of 81 represents 1; mantissa 4000, exponent 81 equals 1.5; mantissa 6000 exponent 81 equals 1.75.

To demonstrate how real numbers are stored, I have included a short BASIC program which will allow you to explore the system. Program 4.3 prints out the five bytes of any floating point number that you care to enter. The values given are in hexadecimal. If you spend some time with the program you should be able to estimate floating point representations and then check your guess!

There are two important points to remember about real numbers. The use of bit 7 of the fourth byte to represent the sign of the value means the mantissa is only a 15-bit value; and zero is treated as a special case. It always has an

exponent and mantissa of 0, but as no other value has an exponent of zero, this is all that needs to be tested to check for a zero value.

Program 4.3

```
100 REM ************************************************************
101 REM **                    Real Numbers Demo                     **
102 REM ************************************************************
110 MODE 1:ZONE 5
120 WHILE 1
130 INPUT "Number";A!
140 PRINT "Number stored as the following:—"
150 PRINT,
160 FOR y= TO 4
170 PRINT HEX$(PEEK(y+@!)),
180 NEXT y
190 PRINT:PRINT "    (____ Mantissa";
200 PRINT "____) Exponent":PRINT
210 WEND
```

SortReal — a floating-point sorting routine

Our new machine code routine has much the same structure as SortInt, in that the outer and inner loops perform similar tasks. Despite the extra complexity involved, SortReal is longer by only 20 or so instructions than SortInt, although some speed potential has been sacrificed in order to keep the size down. When writing a routine, you should always bear in mind the relative importance of speed and size to your particular application.

SortReal (*Source listing 4.2*) gets off to a simple start. BC is loaded with the length of the array from the IX memory area, which is decremented, thus becoming the outer loop counter.

Source code listing 4.2

```
                100 ;SortReal
                110 ;Sorts a BASIC real array into high to low order.
                120 ;Requires the address of the first element and the
                130 ;length of the array to be passed by BASIC.
                140 ;Position independent - corrupts registers AF,BC,DE,HL.
DD4E00          150         LD   C,(IX+0)  ;Get length of array into BC.
DD4601          160         LD   B,(IX+1)
0B              170         DEC  BC
C5              180 PASS    PUSH BC        ;Point HL to MS byte of first
010400          190         LD   BC,4      ;item, and DE to MS byte of
DD6E02          200         LD   L,(IX+2)  ;second item.
DD6603          210         LD   H,(IX+3)
```

```
09        220          ADD  HL,BC
54        230          LD   D,H
5D        240          LD   E,L
09        250          ADD  HL,BC
EB        260          EX   DE,HL
2B        270          DEC  HL
C1        280          POP  BC
C5        290          PUSH BC          ;Save outer loop counter.
C5        300 COMPR    PUSH BC          ;Save inner loop counter.
D5        310          PUSH DE
E5        320          PUSH HL
1A        330          LD   A,(DE)      ;Test for equal signs.
AE        340          XOR  (HL)
17        350          RLA
3004      360          JR   NC,ZERO?    ;If equal jump to ZERO?
1A        370 MOVON    LD   A,(DE)      ;Put upper number's sign into
17        380          RLA   ;           carry and then jump to SWAP?
182B      390          JR   SWAP?
23        400 ZERO?    INC  HL          ;Point to lower number's exponent.
7E        410          LD   A,(HL)      ;Test it.
A7        420          AND  A
28F7      430          JR   Z,MOVON     ;If it is zero, jump to MOVON.
13        440          INC  DE          ;Point to upper number's exponent.
1A        450          LD   A,(DE)      ;Test it.
A7        460          AND  A
2006      470          JR   NZ,EXPO?    ;If it is not zero, jump to EXPO?
2B        480          DEC  HL
7E        490          LD   A,(HL)      ;Put lower number's sign into
17        500          RLA   ;           carry.
3F        510          CCF   ;           Invert it and then
181B      520          JR   SWAP?       ;jump to SWAP?
1A        530 EXPO?    LD   A,(DE)      ;Compare exponents and if not equal
96        540          SUB  (HL)        ;jump to MINU?
2012      550          JR   NZ,MINU?
01FCFF    560          LD   BC,#FFFC    ;Subtract lower number's mantissa
EB        570          EX   DE,HL       ;from upper's.
09        580          ADD  HL,BC
EB        590          EX   DE,HL
09        600          ADD  HL,BC
0604      610          LD   B,4
1A        620          LD   A,(DE)
96        630          SUB  (HL)
23        640 MANLP    INC  HL
13        650          INC  DE
1A        660          LD   A,(DE)
9E        670          SBC  A,(HL)
10FA      680          DJNZ MANLP
17        690          RLA   ;           Put MS bit of result into carry.
2B        700 MINU?    DEC  HL
46        710          LD   B,(HL)      ;If values are negative, complement
1F        720          RRA   ;           the carry flag.
A8        730          XOR  B
17        740          RLA
```

```
E1          750 SWAP?   POP   HL          ;Restore pointers
D1          760         POP   DE
3002        770         JR    NC,NXCM?    ;Test if swap possible.
62          780         LD    H,D         ;If not, point HL to new
6B          790         LD    L,E         ;"lowest" value.
C1          800 NXCM?   POP   BC
0B          810         DEC   BC          ;Test for last comparision.
78          820         LD    A,B
B1          830         OR    C
280A        840         JR    Z,ENDCP     ;If it is jump to ENDCP.
C5          850         PUSH  BC
010500      860         LD    BC,5        ;Point DE to MS byte of next
EB          870         EX    DE,HL       ;item's mantissa.....
09          880         ADD   HL,BC
EB          890         EX    DE,HL
C1          900         POP   BC          ;and then jump back for the
1833        910         JR    COMPR       ;next comparison.
13          920 ENDCP   INC   DE          ;Swap lowest value item found with
23          930         INC   HL          ;last item, even if it is itself!
0605        940         LD    B,5
1A          950 XFER    LD    A,(DE)
4E          960         LD    C,(HL)
77          970         LD    (HL),A
79          980         LD    A,C
12          990         LD    (DE),A
2B         1000         DEC   HL
1B         1010         DEC   DE
10F7       1020         DJNZ  XFER
C1         1030         POP   BC          ;Test if another pass is required.
0B         1040         DEC   BC
78         1050         LD    A,B
B1         1060         OR    C
208E       1070         JR    NZ,PASS
C9         1080         RET
```

The outer loop begins with lines 180-280, which fetch the address of the first item and use this information to point HL to the fourth byte of the first variable and DE to the fourth byte of the second variable. If you refer back to the description of the real number format, you will see that the fourth byte is the MSB of the digits, and also holds the sign of the number in bit 7.
Note that lines 180 and 280 PUSH and POP the outer loop counter so that we can use BC to add 4 to the addresses fetched.
Also examine the use of EX DE,HL; this one-byte, four-cycle instruction is very handy. It can also be used to alleviate the restriction of only being able to perform arithmetic on the HL pair. Because we can quickly swap the values in DE and HL, perform the arithmetic, and then swap back, DE can be used for instructions that normally work exclusively on HL, with only a two-byte, eight-cycle penalty.

The inner loop

Before we begin the comparison section of our routine, we must retrieve the outer loop counter from the stack and immediately re-save it — this loads BC with a value (diminishing on each pass of the outer loop) for use as the inner loop counter (lines 280-290). Before any work begins, we must take the precaution of saving the inner loop counter and our pointers. The inner loop can be considered as two sections — comparison (330-730) and SWAP? (750-790). As soon as a decision has been made about swapping the values, a jump is made to the SWAP? section. Because this jump can be made at a number of different places we cannot predict where HL and DE will be pointing, which is why we save the pointers.

Comparing values

The first step in the comparison section (lines 300-740) is to compare the signs of the values to be compared. This information is held in bit seven of the fourth byte, which is why we begin by pointing there. Lines 330-360 make a good example of the use of an XOR instruction. In Chapter 3 I explained that the result of an XOR is 1 if two bits are different, otherwise the bit will be reset. By XORing the fourth byte of each variable together, bit 7 of the result will be set if the seventh bits of the bytes compared are different from each other, and therefore the two values are of different signs. If they are not, a jump is made to ZERO?. If they are, then a decision can be made. The sign of the second number is placed into the carry and a jump made to SWAP?. The next test, ZERO? is included because of the special case of representing 0 mentioned above. The first value's mantissa is tested: notice the AND instructions which are all that are required to set the flags. If the test does turn out to be zero, then some byte saving is achieved with a jump back to the label MOVON — the same codes are used to slightly different effect. DE is still pointing to the fourth, signed byte of the second value, and the routine will decide if a swap is required on the basis of the sign (a negative number is less than zero). MOVON puts the sign in the carry and jumps to SWAP?
The routine reaches line 440 if the two values have the same sign and the first is not zero. The next test is to see if the second value is zero. Lines 480-500 do this, pointing DE to the exponent and testing for zero. If true, the complement of the first item's sign is put into carry and we jump to SWAP?
Lines 530-550 test for equal exponents; if they are unequal then a decision can be made about the swap. After the SUB instruction, the zero flag will be clear if the exponents are unequal, in which case a jump is made to the label MINU?. Note that carry will be set if the exponent at (HL) is larger than that at (DE), and this information is carried forward as the basis of the SWAP? decision.

Let us recap on what the comparison section has dealt with so far. Values of different signs have caused a jump to SWAP?, as has the discovery that either value is zero. The exponents have been compared, and if they were unequal, a jump to MINU? (which is a precursor to the SWAP? section) has been effected. In all the above cases, the carry flag has been set to indicate which value is the higher.

The last stages, lines 560-690, is only reached if the values are of equal sign and exponent, and are not zero. The pointers in HL and DE are set to point to the LSB of the values. The way we go about this may seem a little odd; it is an example of how saving one byte can cause muddle! There is no SUB instruction for register pairs, and if we use SBC we must clear the carry unless we are sure of its state. An alternative is to use ADD, but add a negative, 16-bit number. Two's complement is not just a convention; it actually works. Adding &FFFC has the same effect as subtracting 4, so we can use ADD and ignore the carry. Making machine code compact does not make it easier to follow!

Subtracting 4 from DE and HL leaves them pointing to the first bytes of each value. The routine now prepares for a loop with lines 610-630. If we were only concerned with speed, it would be slightly quicker not to use a loop for this stage; repeating lines 640-670 an extra three times would make the routine eight bytes longer but save 54 clock cycles for each comparison that reached the final test. I have opted for the slower version, but you might want to try the faster method.

Just before the loop, B is loaded with 4 so as to act as a counter, and the first subtraction performed with SUB. The loop then increments the pointers and uses SBC, so that at the end of the loop the two 4-byte values have been compared, with bit 7 of the accumulator reflecting which was the larger. Remember that the original seventh bits of the fourth bytes indicated the sign, and this stage of the comparison would not have been reached if they had been unequal. As each bit 7 was the same, bit 7 of the result will faithfully reflect any borrow from bit 6 of the subtraction. A borrow would indicate that the value at (HL) is bigger than that at (DE); so bit 7 is rotated into the carry to be used to decide whether a swap is required.

The SWAP? section

This can be entered at one of two points: MINU? or SWAP?. The first (lines 700-740), is reached by jumps from the comparison section that have not tested the sign of the values. The carry tells us which value is the greater, and we know the signs are the same, but the routine must take account of the fact that negative values need to be placed in the reverse order to positive ones: −2 is higher than −3, while +2 is lower than +3.

The sign of the values is transferred from bit 7 of (HL) to bit 7 of B. The carry is then rotated into bit 7 of A. XOR B has the effect of inverting the original value of the carry if the values are negative. The result is then rotated back into the carry.

SWAP? (lines 750-790), having restored the original pointers, is a very simple test, as we have arranged things in such a way that the need to swap items is dictated by the carry. If you wish to change the order of the sort then change line 770 to JR C,NXCM? (38 02).

The delayed replacement principle now comes into effect, because if a swap is not required, the HL pointer is loaded with a new address, that of the lower value — no swap actually occurs until the end of the inner loop. Lines 800-840 are the by now familiar end-of-loop instructions, to which a new twist is added. Because we only want to increment the pointer to the second item if we are going to repeat the loop, line 840 jumps out of the loop when BC reaches zero but before we alter DE. If the loop is to continue, we add 5 to DE, pointing it to the next item, and then perform an unconditional jump to COMPR.

The tail end of the inner loop begins at ENDCP. The lowest item found during the last pass is swapped with the last unsorted item by a loop which has 5 iterations, one for each byte, using C as a temporary buffer. Even if HL is pointing to the same variable as DE, which is to say that the lowest item found was also the last, the swap occurs — the number of times that this is likely to happen is probably quite small, so it is arguable as to whether a check, although easy to program, is worth the extra bytes.

The outer loop

The end of the routine recovers the outer loop counter, decrements and tests it, and jumps back to PASS if the inner loop still needs to be applied to the array.

The most important principle demonstrated by this routine is that the careful use of the flags register can make for economical programming. All the tests are arranged so that the carry flag reflects the order in which the values should be placed. Rotate is very handy for testing the most significant bit of a byte (or indeed the last significant), while XOR can be very powerful in certain circumstances.

CHAPTER 5

Sorting String Arrays

We have sorted integer and real arrays — in this chapter it is the turn of string arrays. The first machine code routine, SortString, will handle single-dimension arrays of character strings. Later we will look at how to sort string arrays which are multi-dimensional.

Program 4.3 gives the new set of data statements for SortString, to be inserted into Program 4.1. Again, you must make a number of changes to the names of the variables, and there is a small challenge to your BASIC programming skills in the creation of random strings, particularly if you make them of random length. Failing this, and it is perhaps a better test of the routine, you can read strings from data statements.

Program 5.1

```
9000 REM ********************************
9001 REM **      SortString Data         **
9002 REM ********************************
9010 DATA 88
9020 DATA DD,4E,00,DD,46,01,0B,C5,799
9030 DATA DD,6E,02,DD,66,03,54,5D,836
9040 DATA 13,13,13,C5,D5,E5,1A,4F,801
9050 DATA 0C,46,04,EB,23,7E,23,66,619
9060 DATA 6F,EB,23,7E,23,66,6F,A7,922
9070 DATA 05,28,0A,3F,0D,28,06,1A,203
9080 DATA 96,13,23,28,F3,E1,D1,38,977
9090 DATA 02,62,6B,C1,0B,78,B1,28,748
9100 DATA 05,13,13,13,18,CD,06,03,300
9110 DATA 1A,4F,7E,12,71,13,23,10,432
9120 DATA F7,C1,0B,78,B1,20,B0,C9,1157
```

String arrays in BASIC

Before looking at *Source Listing 5.1*, we need to know how BASIC strings are stored in an array. Each dimensioned string array has its own string descriptor block, holding three bytes for each element. The first byte gives the length of

Source code listing 5.1

```
                100 ;SortString
                110 ;Sorts a BASIC single dimensioned string array into
                120 ;alphabetic order.
                130 ;Requires the address of the first element and the
                140 ;length of the array to be passed by BASIC.
                150 ;Position independent  - corrupts registers AF,BC,DE,HL.
DD4E00          160         LD   C,(IX+0)  ;Get length of array into BC.
DD4601          170         LD   B,(IX+1)
0B              180         DEC  BC
C5              190 PASS    PUSH BC        ;Save outer loop counter.
DD6E02          200         LD   L,(IX+2)  ;Point HL to first string
DD6603          210         LD   H,(IX+3)  ;descriptor
54              220         LD   D,H       ;Point DE to second string
5D              230         LD   E,L       ;descriptor.
13              240         INC  DE
13              250         INC  DE
13              260         INC  DE
C5              270 COMPR   PUSH BC        ;Save inner loop counter.
D5              280         PUSH DE
E5              290         PUSH HL
1A              300         LD   A,(DE)    ;Load C with length of upper string.
4F              310         LD   C,A
0C              320         INC  C
46              330         LD   B,(HL)    ;Load B with length of lower string.
04              340         INC  B
EB              350         EX   DE,HL
23              360         INC  HL        ;Point DE to upper string.
7E              370         LD   A,(HL)
23              380         INC  HL
66              390         LD   H,(HL)
6F              400         LD   L,A
EB              410         EX   DE,HL
23              420         INC  HL        ;Point HL to lower string.
7E              430         LD   A,(HL)
23              440         INC  HL
66              450         LD   H,(HL)
6F              460         LD   L,A
A7              470         AND  A         ;Clear carry flag.
05              480 CLOOP   DEC  B         ;Decrement lengths, and if either
280A            490         JR   Z,SWAP?   ;reach zero, set the carry flag
3F              500         CCF  ;          accordingly and jump to SWAP?
0D              510         DEC  C
2806            520         JR   Z,SWAP?
1A              530         LD   A,(DE)    ;Compare characters from strings,
96              540         SUB  (HL)      ;increment string pointers, and if
13              550         INC  DE        ;the characters were equal, jump
23              560         INC  HL        ;back to CLOOP.
28F3            570         JR   Z,CLOOP
E1              580 SWAP?   POP  HL        ;Restore pointers
D1              590         POP  DE
3802            600         JR   C,NXCM?   ;Test if swap possible.
62              610         LD   H,D       ;If not, point HL to new
```

```
6B          620          LD   L,E        ;"lowest" string.
C1          630 NXCM?    POP  BC
0B          640          DEC  BC         ;Test for last comparision.
78          650          LD   A,B
B1          660          OR   C
2805        670          JR   Z,ENDCP    ;If it is jump to ENDCP.
13          680          INC  DE         ;Point to next string and jump back
13          690          INC  DE         ;for the next comparision.
13          700          INC  DE
18CD        710          JR   COMPR
0603        720 ENDCP    LD   B,3
1A          730 XFDSC    LD   A,(DE)
4F          740          LD   C,A
7E          750          LD   A,(HL)
12          760          LD   (DE),A
71          770          LD   (HL),C
13          780          INC  DE
23          790          INC  HL
10F7        800          DJNZ XFDSC
C1          810          POP  BC         ;Test if another pass is required.
0B          820          DEC  BC
78          830          LD   A,B
B1          840          OR   C
20B0        850          JR   NZ,PASS
C9          860          RET
```

the string, which explains why BASIC strings can only be 255 characters in length. Bytes two and three hold, in Z80 notation, the address of the start of the string. The actual location of the string will depend on how it was entered — you will find that strings read from data statements will have a descriptor pointing to the BASIC program where the string is held. The practical upshot of all this is that to find a string we look up the address in its descriptor block, and if we want to swap two strings over we only need to swap the three bytes of their respective descriptors.

In a single-dimension array, consisting of say 10 items, the descriptor block will be 30 bytes long. We can find the block with the @ prefix — this will pass the address to our machine code routine. The fact that the strings themselves may be dotted all over RAM is no problem; we can find each one as we need it from its address in the block, and we don't need to change any of the strings themselves. We have all the ingredients to write a sorting routine.

SortString in action

SortString uses the delayed replacement method and is very similar in principle to SortReal. The two essential differences are the comparison section and the simpler code required to swap items — we swap only the descriptors.

The routine begins the outer loop by fetching the length of the string and setting up BC as the outer loop counter. HL is pointed to the first and DE to the second of the descriptors in the array block, and then the inner loop begins. After pushing the register pairs, the lengths of the two strings are placed in B and C, and we reload HL and DE to point not to the descriptors but to the strings themselves.

CLOOP (representing Comparison LOOP, lines 480-570) has three exit points. This is necessary because we need to take the string lengths into acount — ABC must be placed before ABCD. As we examine the individual characters of both strings we will probably find a pair that are not the same; this is one basis of a comparison and forms one of the exits from the loop (line 570). If this does not happen before we reach the end of one of the strings, then the lengths dictate the final order. B and C are used to hold the number of characters left to test, so they are decremented on each pass of CLOOP. The first instructions (lines 480-520) test to see if either has reached zero and exits will occur at lines 490 or 520 if this is the case.

We use the carry to convey information to the next section of the routine. The flag is cleared by line 470; remember that DEC r instructions do not affect the carry, so an exit at line 490 will take a clear carry flag with it. CCF (complement carry flag) is used to set the flag and the counter in register C is decremented. If this causes an exit at line 570 then a set carry flag is passed on.

If the first two tests show that there are characters to be compared (for example, if the first pass of CLOOP revealed that both strings consisted of at least one character), then lines 530-540 are reached. Using subtraction, these compare the ASCII codes of the characters we are currently pointing to — zero will result if they are equal. Before testing for a match, the pointers are incremented so as to point to the next potential characters in the two strings. Now the conditional jump instruction can decide whether the ASCII codes were the same — if they were then we will need to look at further characters (if there are any), so a jump back for another pass is made. Note that if this happens, the carry will be clear for the first stage of CLOOP. If Z is not set, then the characters are different: we pass on to the next stage with carry indicating which character had the higher ASCII code.

SWAP?

Before a decision is made as to whether a swap is required, the addresses of the descriptors are retrieved from the stack. Whichever exit was taken from CLOOP, the carry flag will be set if the second string should be placed before the first, which means we need to swap the descriptors. If carry is not set, HL is loaded with the address of the new 'lowest found so far' string.

Lines 630-670 are a very straightforward ending to the inner loop, pointing DE to the next descriptor before jumping back if further comparisons are to be made. The outer loop ends by swapping the lowest string descriptor with the last of the unsorted block; it uses a simple three-pass loop to save bytes. The outer loop counter is then salvaged from the stack and the decision made as to whether a further pass is required.

Using SortString to order an array in Z to A order is possible because of the manner in which CLOOP works — just alter line 600 to JR NC,NXCMP? (30 02).

The three sorting routines described so far in this book are simple and fast methods of sorting arrays of each type of variable available in Locomotive BASIC. Not only can they be included in BASIC programs; they can also be called from other machine code programs, and even modified to sort other areas of data. The CLOOP section of SortString can be lifted out and used as a 'string compare' subroutine for any two ASCII strings provided you match the entry requirements: (B and C with the lengths, DE and HL pointing to the first character of each string).

SortArray — the heart of a database program

The next routine, SortArray, is written for a more specific application, while still demonstrating sorting principles. Its purpose is to give database programs a powerful facility by sorting a two-dimensional string array according to the contents of one of its dimensions.

SortArray can also be instructed to work in either direction. Additionally, simple number handling allows it to sort positive integer values stored as strings in one dimension of the array used for the database (BASE$).

Database (Program 5.2) is the type of program that can be of great use for home or business purposes. At its simplest level it simulates a file card system, with the added bonus of print-outs, and it has the ability efficiently to sort and

Program 5.2

```
1000 REM ******************************
1001 REM **     Database Program     **
1002 REM ******************************
1003 REM **  After a break or error  **
1004 REM **  GOTO 1500 to save data  **
1005 REM ******************************
1010 ZONE 18:MODE 2
1020 WINDOW #0,1,80,3,23
1030 WINDOW #1,1,80,1,1
1040 WINDOW #2,1,80,25,25
1050 PRINT #1,TAB(29);"*** Database Program ***"
1060 GOSUB 8000
```

```
1070 MAXI%=250
1080 INPUT #2,"Load a database (Y/N)";T$
1090 IF T$="Y" OR T$="y" THEN GOSUB 2040:GOTO 1500
1100 REM ******************************
1101 REM **   Set Up New Database   **
1102 REM ******************************
1110 PRINT #1,TAB(27)"*** Set Up New Database ***"
1120 INPUT #2,"How many fields do you require";F%
1130 F%=F%-1:IF F%>255 OR F%<1 THEN 1120
1140 DIM BASE$(MAXI%,F%),NM$(F%)
1150 FOR X=0 TO F%
1160  PRINT"Field ";X+1;" Title -",
1170  PRINT #2,"Title?",:LINE INPUT £2,NM$(X)
1180  PRINT NM$(X)
1190 NEXT X
1200 IM%=-1:GOSUB 5000
1500 REM ******************************
1501 REM **          Menu            **
1502 REM ******************************
1510 WHILE 1
1520  RESTORE 10010
1530  PRINT #1,TAB(28);"********  Menu  ********"
1540  CLS
1550  FOR X=1 TO 6
1560   READ P$
1570   PRINT CHR$(10);CHR$(10),P$,,"-----";X
1580  NEXT X
1590  PRINT #2,TAB(23);"Press the required function number"
1600  M$=INKEY$:IF M$<"1" OR M$>"6" THEN 1600
1610  CLS
1620  ON VAL(M$) GOSUB 2000,3000,4000,5000,6000,7000
1630 WEND
2000 REM ******************************
2001 REM **        Load Data         **
2002 REM ******************************
2010 INPUT #2,"Delete old database (Y/N)";Q$
2020 IF Q$<>"Y" AND Q$<>"y" THEN RETURN
2030 ERASE NM$,BASE$
2040 PRINT #1,TAB(28);"***** Loading Data *****"
2050 INPUT #2,"Name of database";D$
2060 IF D$="" OR LEN(D$)>8 THEN 2050
2070 CLS #2:LOCATE 1,23
2080 OPENIN D$
2090 INPUT #9,IM%,F%
2090 INPUT £9,IM%,F%
2100 DIM NM$(F%),BASE$(MAXI%,F%)
2110 FOR X=0 TO F%
2120  INPUT #9,NM$(X)
2130 NEXT X
```

```
2140 FOR X=0 TO IM%
2150  FOR Y=0 TO F%
2160   INPUT #9,BASE$(X,Y)
2170  NEXT Y
2180 NEXT X
2190 CLOSEIN
2200 RETURN
3000 REM ******************************
3001 REM **       Save Data          **
3002 REM ******************************
3010 PRINT #1,TAB(28);"***** Saving  Data *****"
3020 INPUT #2,"Name of data to be saved";D$
3030 IF D$="" OR LEN(D$)>8 THEN 3020
3040 CLS #2:LOCATE 1,23
3050 OPENOUT D$:WRITE #9,IM%,F%
3060 FOR X=0 TO F%
3070  WRITE #9,NM$(X)
3080 NEXT X
3090 FOR X=0 TO IM%
3100  FOR Y=0 TO F%
3110   WRITE #9,BASE$(X,Y)
3120  NEXT Y
3130 NEXT X
3140 CLOSEOUT
3150 RETURN
4000 REM ******************************
4001 REM **      Examine Files       **
4002 REM ******************************
4010 PRINT #1,TAB(28);"***** Examine Data *****"
4020 INPUT #2,"Which file number";EX%:EX%=EX%-1
4030 PRINT #2,CHR$(243);" Next | ";CHR$(242);" Back | ";
4040 PRINT #2,CHR$(241);" 8 Ahead |↑ 8 Back | ";
4050 PRINT #2,"COPY - Edit | ENTER - Menu | S - Search";
4060 IF EX%<0 THEN EX%=0
4070 IF EX%>=IM% THEN EX%=IM%
4080 C%=0:GOSUB 7500
4090 T$=INKEY$:IF T$="" THEN 4090
4100 T%=ASC(T$)
4110 IF T%=13 THEN RETURN
4120 IF T%=224 THEN 4190
4130 IF T%<240 OR T>243 THEN 4090
4140 IF T%=240 THEN EX%=EX%-8
4150 IF T%=241 THEN EX%=EX%+8
4160 IF T%=242 THEN EX%=EX%-1
4170 IF T%=243 THEN EX%=EX%+1
4180 GOTO 4060
4190 PRINT #1,TAB(28);"***** Editing File *****"
4200 INPUT #2,"Which field do you wish to alter";T%:T%=T%-1
4210 IF T%<0 OR T%>F% THEN 4200
```

```
4220 PRINT #2,"Entry?",:LINE INPUT #2,BASE$(EX%,T%)
4230 C%=0:GOSUB 7500
4240 INPUT #2,"Another field? (Y/N)";T$
4250 IF T$="Y" OR T$="y" THEN 4200
4260 PRINT #1,TAB(28);"***** Examine Data *****"
4270 GOTO 4030
5000 REM ****************************
5001 REM **        Add Files         **
5002 REM ****************************
5010 PRINT #1,TAB(28);"***** Adding Files *****"
5020 MORE=1
5030 WHILE MORE
5040 IM%=IM%+1
5050  CLS:PRINT"File No. ";IM%+1
5060  FOR X=0 TO F%
5070   PRINT X+1;" ";NM$(X);" -",
5080   PRINT #2,"Entry?",
5090   LINE INPUT #2,BASE$(IM%,X)
5100   PRINT BASE$(IM%,X)
5110  NEXT X
5120  IF IM%>=MAXI% THEN MORE=0
5130  INPUT #2,"Another file (Y/N)";T$
5140  IF T$="N" OR T$="n" THEN MORE=0
5150 WEND
5160 RETURN
6000 REM ****************************
6001 REM **        Sort  Data        **
6002 REM ****************************
6010 PRINT #1,TAB(28);"****** Sort  Data ******"
6020 INPUT #2,"By which field do you wish to sort?",S%
6030 S%=S%-1:IF S%<0 OR S%>F% THEN 6020
6040 INPUT #2,"Do you wish to sort A to Z? (Y/N)";T$
6050 IF T$="N" OR T$="n" THEN DR%=1 ELSE DR%=0
6060 PRINT #1,TAB(28);"******** Sorting *******"
6070 CALL HIMEM+1,@BASE$(0,S%),IM%,F%,@BASE$(0,1),@BASE$(0,0),DR%
6080 RETURN
7000 REM ****************************
7001 REM **        Print Out         **
7002 REM ****************************
7010 PRINT #1,TAB(28);"****** Print  Out ******"
7020 INPUT #2,"From file no.";ST
7030 ST=ST-1:IF ST<1 THEN ST=0
7040 INPUT #2,"To file no.";ED
7050 ED=ED-1:IF ED>IM% THEN ED=IM%
7060 C%=8
7070 FOR W=ST TO ED
7080  EX%=W
7090  GOSUB 7500
7100 NEXT W
```

```
7110 RETURN
7500 REM ******************************
7501 REM **       Print File         **
7502 REM ******************************
7510 CLS
7520 PRINT #C%,"File No.";EX%+1
7530 FOR X=0 TO F%
7540  PRINT #C%,X+1;" ";NM$(X);" -",BASE$(EX%,X)
7550 NEXT X
7560 RETURN
9000 REM ******************************
9001 REM **      SortArray Data      **
9002 REM ******************************
9010 DATA 232
9020 DATA DD,5E,02,DD,56,03,DD,6E,958
9030 DATA 04,DD,66,05,A7,ED,52,DD,1039
9040 DATA 75,04,DD,74,05,DD,4E,08,770
9050 DATA DD,46,09,78,B1,C8,C5,DD,1215
9060 DATA 6E,0A,DD,66,0B,54,5D,13,650
9070 DATA 13,13,C5,D5,E5,1A,A7,28,910
9080 DATA 5C,4F,0C,7E,A7,3F,28,55,664
9090 DATA 47,04,EB,23,7E,23,66,6F,719
9100 DATA EB,23,7E,23,66,6F,7E,D6,984
9110 DATA 30,D6,0A,30,2F,1A,D6,30,655
9120 DATA D6,0A,30,28,E5,D5,C5,0E,965
9130 DATA 00,7E,D6,30,D6,0A,30,04,664
9140 DATA 0C,23,10,F5,61,C1,C5,41,860
9150 DATA 0E,00,1A,D6,30,D6,0A,30,574
9160 DATA 04,0C,13,10,F5,79,94,C1,758
9170 DATA D1,E1,20,11,05,28,0E,3F,605
9180 DATA 0D,28,0A,1A,96,13,23,28,333
9190 DATA F3,18,02,18,91,E1,D1,17,895
9200 DATA DD,AE,00,1F,38,02,62,6B,689
9210 DATA C1,0B,78,B1,28,05,13,13,584
9220 DATA 13,18,87,DD,4E,0A,DD,46,778
9230 DATA 0B,A7,ED,42,EB,A7,ED,42,1186
9240 DATA DD,4E,02,DD,46,03,09,EB,839
9250 DATA 09,DD,46,06,04,C5,D5,E5,949
9260 DATA 06,03,1A,4F,7E,12,71,13,390
9270 DATA 23,10,F7,E1,D1,DD,4E,04,1035
9280 DATA DD,46,05,EB,09,EB,09,C1,977
9290 DATA 10,E3,C1,0B,78,B1,20,AB,947
9300 DATA C9,00,00,00,00,00,00,00,201
10000 REM ******************************
10001 REM **         Menu Data        **
10002 REM ******************************
10010 DATA Load data,Save data,Examine fi
10020 DATA Add data,Sort data,Print data
```

sift through the information it contains. You can use it as an address book which can list all your contacts that live in any particular town or city. You can use it to catalogue a stamp collection by country of origin, and, in a few moments, re-list the stamps in order of age.

Before we launch into the program let me define the terms that I shall use to describe the database. The information is stored in a two dimensional string array. One dimension of the array has as many elements as there are to be 'fields'; these are the various subheadings such as name, address, age and so on. Therefore, one field will perhaps contain all the surnames of the people in your database array.

The other dimension has as many elements as there are 'files'; each file contains all the fields pertaining to an individual record. One file of the array will contain all the fields — name, address etc. — relevant to a particular person (or perhaps cassette, stamp or whatever you are keeping a record of). Each element is a string of up to 255 characters, each having both a file and field number. Elements with the same field number contain the same type of data, and elements with the same file number relate to the same record.

The bulk of the program is written in BASIC for two reasons: you will be able to expand and modify it to meet your personal requirements with ease, and Locomotive BASIC is fast enough for most of the program to be useful without resorting to machine code. As I hope you will customise the program to suit you, the following is a commentary on the BASIC.

Database

Lines 1000-1070: Set up the screen display, load the machine code and set MAXI% to 250. This sets the maximum number of files that the program will hold — you may specify a much larger value if you are likely to use short strings or only a few fields.

Lines 1080-1090: Give the opportunity for loading a new set of information from the keyboard, or loading an old database from tape (or disk).

Lines 1120-1140: Ask how many fields are needed and dimension the main and title array accordingly. Note that the first field is numbered 0, so line 1130 subtracts 1 from F%.

Lines 1150-1190: A loop that inputs the title you want for each field — it will tidy up screen display if you keep the titles reasonably short.

Line 1220: IM% is the number of the last file in BASE$ that holds data. The numbering begins at 0, so −1 is an invalid file. At this point execution is

passed to a subroutine which is also available from the menu (Add data).

Lines 1500-1630: A closed loop to print the menu from data steatements, fetch a valid choice, and GOSUB the required function.

Lines 2000-2170: Fetches a database from a suitable tape or disk file. IM% and F% are read in, new arrays are dimensioned and filled with information from tape or disk.

Lines 3000-3150: Dumps the current database to tape or disk in a form suitable for the LOAD subroutine to read.

Lines 4000-4080: The start of the Examine routine asks which file you wish to begin with (EX%), prints the prompting messages and then uses the Print subroutine to display the first file.

Lines 4090-4100: Fetches a valid response to the prompts.

Lines 4110: If the response is ENTER, return to menu.

Line 4120: If the response is COPY then jump to the edit section.

Lines 4130-4180: Other valid responses change EX% as required and then jump back to 4060, where EX% is made legal, and the new file displayed.

Lines 4190-4270: Gives the chance to change any field of the current file.

Lines 5000-5160: New files are added by eliciting each field from the keyboard. IM% is incremented and compared with MAXI% to ensure we do not overrun the size of the array.

Lines 6000-6080: The subroutine to call SortArray. Note the parameters of the CALL command. It is possible to rewrite this section to allow you to sort only, say, the first 10 files.

Lines 7000-7110: A rudimentary printer driver. It uses the print subroutine. You can expand this subroutine (if you have a printer!) to be more selective.

Lines 7500—7560: This subroutine is called from three places in the program: the examine, print and edit options. It sends the contents of the file number EX% to the channel defined by C%.

Lines 8000-8190: The code loading routine. This. and the following section, can be replaced with a loading operation from tape or disk.

Lines 9000-9280: SortArray in data statements.

Lines 10000 on: The function names for the menu.

Database makes good use of the Amstrad's windowing functions. Two single-line windows at the top and bottom of the screen are used for titles and prompts. Notice that the program always inputs to stream 2, the bottom window; and an oddity of Locomotive BASIC (at least on my CPC 464) means that string prompts included in LINE INPUT commands appear on stream 0, whichever channel you specify. As a result, separate PRINT #2 commands have to be used: if your machine behaves in a more predictable manner then you can save some lines of BASIC.
Using the program is quite simple. The main caution I will give is do plan the number and the titles of fields with care. For example, split names into fore and surnames, and addresses by road, town, etc; this will give many more options when you want to sort data. Numbers may require special treatment, as I will explain when we look at the machine code. Don't be afraid to declare a few extra fields for comments and future expansion — you can give these new names by breaking into BASIC and redefining the appropriate element of TITLE (NM$).
You may notice that the prompt line of the examine subroutine gives a search option that does not, as yet, work. The next chapter will give a further machine-code routine to implement this option, with the appropriate BASIC additions.

Parameters for SortArray

SortArray uses many more parameters than our previous routines. In the order that they appear above the IX pointer (and therefore the reverse order to that in which they must be declared), they are as follows:

IX+0 and IX+1: the direction of the sort. Bit 0 of (IX+0) is all that is used by the machine code. Passing an even number will initiate an A to Z sort; conversely an odd value forces a Z to A result.

IX+2 and IX+3: the address of element (0,0) of the array.

IX+4 and IX+5: the address of element (0,1) of the array.

IX+6 and IX+7: the number of fields. An eight-bit number is all that can be handled, so 256 is the maximum number of fields possible. There is no reason why SortArray could not handle more fields, but as this seems an unlikely requirement, the restriction allows SortArray to use eight-bit loop counters. Note that fields are numbered in the range 0-255 for the purposes of the machine code and array numbering, although BASIC translates the range to 1-256 for display purposes.

IX+8 and IX+9: the number of files (less one). Note that this need not be the same as the total number of elements dimensioned.

IX+10 and IX+11: the address of the first element in the field on which you wish to sort.

It is worth mentioning that the field and file variables in *Database* (F% and IM%) begin at zero, and the machine code takes this into account.

SortArray in action

Source code listing 5.2 shows SortArray. The same structure as our previous DR sorts is used — outer loop and inner loop. CLOOP, from SortString, is also evident. In fact, the main body of the routine is very similar to SortString, with the added complication of handling strings as numbers.

Source code listing 5.2

```
            100 ;SortArray
            110 ;Sorts a BASIC multi-dimensioned string array into
            120 ;alphabetic order, or if both strings begin with a
            130 ;numerical character, arithmetic order.
            140 ;Does not handle negative numbers.
            150 ;Direction of sort determined by an entry parameter.
            160 ;Addresses of field which determines sort, first and
            170 ;second field, number of fields and items and direction
            180 ;of sort to be passed by BASIC.
            190 ;Position independent  - corrupts registers AF,BC,DE,HL
DD5E02      200         LD   E,(IX+2)  ;Calculate the distance between
DD5603      210         LD   D,(IX+3)  ;fields and store the result in
DD6E04      220         LD   L,(IX+4)  ;memory at IX relative addresses.
DD6605      230         LD   H,(IX+5)
A7          240         AND  A
ED52        250         SBC  HL,DE
DD7504      260         LD   (IX+4),L
DD7405      270         LD   (IX+5),H
DD4E08      280         LD   C,(IX+8)  ;Get number of items into BC.
DD4609      290         LD   B,(IX+9)
78          300         LD   A,B
```

```
B1          310          OR   C
C8          320          RET  Z
C5          330 PASS     PUSH BC          ;Save outer loop counter.
DD6E0A      340          LD   L,(IX+10)   ;Point HL to first string
DD660B      350          LD   H,(IX+11)   ;descriptor
54          360          LD   D,H         ;Point DE to second string
5D          370          LD   E,L         ;descriptor.
13          380          INC  DE
13          390          INC  DE
13          400          INC  DE
C5          410 COMPR    PUSH BC          ;Save inner loop counter.
D5          420          PUSH DE
E5          430          PUSH HL
1A          440          LD   A,(DE)      ;Test upper string length.
A7          450          AND  A
285C        460          JR   Z,SWAP?     ;If empty jump with carry clear.
4F          470          LD   C,A         ;Load C with upper string length.
0C          480          INC  C
7E          490          LD   A,(HL)      ;Test length of lower string.
A7          500          AND  A
3F          510          CCF
2855        520          JR   Z,SWAP?     ;If empty jump with carry set.
47          530          LD   B,A         ;Load B with lower string length.
04          540          INC  B
EB          550          EX   DE,HL
23          560          INC  HL          ;Point DE to upper string.
7E          570          LD   A,(HL)
23          580          INC  HL
66          590          LD   H,(HL)
6F          600          LD   L,A
EB          610          EX   DE,HL
23          620          INC  HL          ;Point HL to lower string.
7E          630          LD   A,(HL)
23          640          INC  HL
66          650          LD   H,(HL)
6F          660          LD   L,A
7E          670          LD   A,(HL)      ;If lower string does not start
D630        680          SUB  £30         ;with a number, jump to CLOOP.
D60A        690          SUB  £0A
302F        700          JR   NC,CLOOP
1A          710          LD   A,(DE)      ;If upper string does not begin
D630        720          SUB  £30         ;with a number, jump to CLOOP.
D60A        730          SUB  £0A
3028        740          JR   NC,CLOOP
E5          750          PUSH HL          ;Save values.
D5          760          PUSH DE
C5          770          PUSH BC
0E00        780          LD   C,0
7E          790 SGLP1    LD   A,(HL)      ;This loop counts the number of
D630        800          SUB  £30         ;numerical characters before a
D60A        810          SUB  £0A         ;decimal point (or other character)
3004        820          JR   NC,SG1DN    ;in the lower string, placing the
0C          830          INC  C           ;result in H.
```

```
23          840          INC  HL
10F5        850          DJNZ SGLP1
61          860 SG1DN    LD   H,C
C1          870          POP  BC          ;Retrieve length of second string.
C5          880          PUSH BC
41          890          LD   B,C         ;Performs same function for upper
0E00        900          LD   C,0         ;string, leaving the result in C.
1A          910 SGLP2    LD   A,(DE)
D630        920          SUB  £30
D60A        930          SUB  £0A
3004        940          JR   NC,SG2DN
0C          950          INC  C
13          960          INC  DE
10F5        970          DJNZ SGLP2
79          980 SG2DN    LD   A,C         ;Test for same sized numbers,
94          990          SUB  H           ;setting carry in the process.
C1         1000          POP  BC          ;Restore old values.
D1         1010          POP  DE
E1         1020          POP  HL
2011       1030          JR   NZ,SWAP?    ;If numbers are not same size, jump.
05         1040 CLOOP    DEC  B           ;Decrement lengths, and if either
280E       1050          JR   Z,SWAP?     ;reach zero, set the carry flag
3F         1060          CCF  ;            accordingly and jump to SWAP?
0D         1080          DEC  C
280A       1090          JR   Z,SWAP?
1A         1100          LD   A,(DE)      ;Compare characters from strings,
96         1110          SUB  (HL)        ;increment string pointers, and if
13         1120          INC  DE          ;the characters were equal, jump
23         1130          INC  HL          ;back to CLOOP.
28F3       1140          JR   Z,CLOOP
1802       1144          JR   SWAP?       ;A "bodge" to allow the routine
1891       1146 PATCH    JR   PASS        ;to be position independent.
E1         1170 SWAP?    POP  HL          ;Restore pointers
D1         1180          POP  DE
17         1190          RLA
DDAE00     1200          XOR  (IX+0)
1F         1210          RRA
3802       1220          JR   C,NXCM?     ;Test if swap possible.
62         1230          LD   H,D         ;If not, point HL to new
6B         1240          LD   L,E         ;"lowest" string.
C1         1250 NXCM?    POP  BC
0B         1260          DEC  BC          ;Test for last comparison.
78         1270          LD   A,B
B1         1280          OR   C
2805       1290          JR   Z,ENDCP     ;If it is jump to ENDCP.
13         1300          INC  DE          ;Point DE to next string descriptor.
13         1310          INC  DE
13         1320          INC  DE
1887       1330          JR   COMPR       ;Jump back for next comparison.
DD4E0A     1340 ENDCP    LD   C,(IX+10)   ;This section points DE and HL
DD460B     1350          LD   B,(IX+11)   ;to the two string descriptors
A7         1360          AND  A           ;to be swapped in the first field.
ED42       1370          SBC  HL,BC
```

```
EB        1380         EX   DE,HL
A7        1390         AND  A
ED42      1400         SBC  HL,BC
DD4E02    1410         LD   C,(IX+2)
DD4603    1420         LD   B,(IX+3)
09        1430         ADD  HL,BC
EB        1440         EX   DE,HL
09        1450         ADD  HL,BC
DD4606    1460         LD   B,(IX+6)   ;Load B with the number of fields.
04        1470         INC  B
C5        1480 XFELM   PUSH BC         ;This loop swaps over the string
D5        1490         PUSH DE         ;descriptors in each field.
E5        1500         PUSH HL
0603      1510         LD   B,3
1A        1520 XFDSC   LD   A,(DE)
4F        1530         LD   C,A
7E        1540         LD   A,(HL)
12        1550         LD   (DE),A
71        1560         LD   (HL),C
13        1570         INC  DE
23        1580         INC  HL
10F7      1590         DJNZ XFDSC
E1        1600         POP  HL
D1        1610         POP  DE
DD4E04    1620         LD   C,(IX+4)   ;Retrieve the pre-calculated
DD4605    1630         LD   B,(IX+5)   ;distance between fields.
EB        1640         EX   DE,HL
09        1650         ADD  HL,BC
EB        1660         EX   DE,HL
09        1670         ADD  HL,BC
C1        1680         POP  BC
10E3      1690         DJNZ XFELM
C1        1700         POP  BC         ;Test if another pass is required.
0B        1710         DEC  BC
78        1720         LD   A,B
B1        1730         OR   C
20AB      1740         JR   NZ,PATCH   ;If it is, jump to PASS via PATCH.
C9        1750         RET
```

Lines 200-270 use RAM locations for the first time to store a routine variable. The array descriptor block is stored in the order of the strings of the first field, followed by the second, and so on. To calculate the distance between fields, the address of element (0,1) is subtracted from the address of element (0,0), and the result placed somewhere that it will be safe until we return to BASIC. We have finished with the RAM locations (IX+4) and IX+5) used by BASIC to pass the address of the second element of the array, so they are available as a data area.

The next stage (lines 280-320) fetches the number of items for use as the loop counters, and we perform a little safety check. In the unlikely event of BASIC passing zero by mistake, the outer loop would be performed 64K times,

involving every byte of RAM! This happened on one occasion while I was still developing the BASIC, so I have left the error check (testing BC for zero) in the routine to help you avoid the same mistake.

Comparison section

You should recognise lines 330-670 as being similar to SortString, with a check for empty strings. Next comes the number handling (lines 680-1030). The lower string's first character is tested to see if it is an ASCII digit, that is, has a value in the range &30 to &39. We test for this with two successive subtractions: subtracting &30 will, if the character is a digit, bring it into the range 0 to 9; a further subtraction of &0A (10) will therefore only cause a carry if the character is in this range.

The second string is tested in the same manner, and if either string does not begin with a digit, then both strings are compared as if they are character strings rather than values by jumping ahead to CLOOP.

Comparing strings as 'numbers'

The next section deals with strings which the routine has decided are numbers; both strings of the comparison begin with digits. Before two loops are used to count the number of digits in each string, the values in BC, DE and HL are saved on the stack.

SGLP1 (lines 790-850) counts the number of significant digits in the string, that is, the number of ASCII digits before a non-digit character. Comparing 2 with 1000 using CLOOP would give 2 a higher significance than 1000 as its first character is higher; to avoid this problem we check through each character of the first string, using C as a counter. On each pass through the loop, if a valid digit is found, C is incremented; the pointer to the string, HL, is also incremented: and then B (holding the total length of the string) is decremented and tested. We will only exit from SGLP1 when we have found a non-numeric character or the end of the string has been reached — in both cases C will hold a value for the number of significant digits in the first string.

When we move on to SGLP2 we will want to retain the value in C, but as HL is now temporarily available, we can move the value into H. BC is now free, allowing us to load C with the length of the second string by POPing and then rePUSHing BC. SGLP2 performs the same task as the first significance loop, except on the second string. By the time we leave the second loop we have two values, in C and H, which we can compare.

Notice that SGLP2 uses the DJNZ instruction even though it means transferring the contents of C to B. If we used DEC C and JR NZ instead, the number of program bytes would remain the same, and even in the case of one

pass of the loop we would only save one machine cycle, because single-byte LD and DEC operations take 4 cycles, and conditional JRs use 13 or 8 cycles depending upon whether the condition is met or not, while DJNZ takes 16 or 11 cycles respectively. Two passes of the loop and we break even, and each pass after that would cost a cycle if we stuck with C. This may seem a rather pedantic point to make, but these are the sort of principles that we will need to bear in mind when writing code to achieve speed — in this case we need not even sacrifice RAM locations.

Back to the listing, and SG2DN (SiGnificance loop 2 DoNe): at this stage the two significant figure values, i.e. the number of digits before a decimal point (or other non-numeric character) are compared. If they are equal and the Z flag is therefore set by the subtraction, we must continue to compare the strings. If they are numbers with the same quantity of significant digits then CLOOP will be able to handle them. On the other hand, if they are not equal, we have sufficient information (in the carry flag, of course) to move on to SWAP?

Ordinary strings reach CLOOP from lines 700 or 740. This is exactly the same as the comparison loop in SortString, with one minor oddity: all routes to CLOOP arrive with carry clear, so no AND A instruction is needed. What are lines 1144-1146 doing in the middle of the routine? SortArray is rather long, over the 128-byte limit that JR can dictate. If we used a JP at the end of the second loop to jump to the beginning, the routine would not be position-independent as we would need to supply an actual address for JP to load into the PC. In order to keep the routine position-independent, the end of the outer loop performs a JR to PATCH, which causes another relative jump to the start of the loop. This is ugly and unstructured but it works. The uneven line numbering helps to show that this is not part of the program flow.

The swap section

SWAP? (lines 1170-1240), having restored the pointers, makes the decision as to whether to move the lowest-string pointer, HL, on the basis of the result in the carry flag, but remember that we want to make SortArray bi-directional. This can be achieved by inverting the carry if a sort is desired in Z to A order, which means we actually use HL as a highest-string pointer. Line 1190 saves the carry in bit 0 of the A register, and then in line 1200 we XOR it with bit 0 of (IX+0), the direction parameter. If this was set, the resultant bit 0 of A is an inversion of the carry, otherwise it holds the original value. The new value is rotated back into carry for use as the basis of the SWAP? decision. This is quite a powerful use of XOR.

Lines 1250-1330 end the inner loop, as seen in SortString.

Now to the part where the whole array is sorted on one field. The address of

the first element in the sorting field is fetched from the parameter area and placed in BC. Lines 1360-1400 subtract this from the two pointers so that HL and DE contain the offset from the beginning of the field of the two descriptors in which we are interested. Note that the first and second string pointers have temporarily changed register pairs due to the EX instruction. We now add these offsets to the base address of the whole array, swapping the registers with another EX. Fetching the number of fields and incrementing it leaves BC as a counter for the next loop, XFELM (transFer ELeMents).

XFDSC is a loop inside this, and swaps the three bytes of each descriptor. When the first field is done, the pre-calculated offset to the next field is added to the original pointers, safely recovered from the stack, and the next field swapped about. The process continues until line 1690 detects that no more fields are to be swapped.

The outer loop ends with its counter being retrieved and decremented. If another pass is required, it is jumped to via PATCH, as described earlier.

SortArray is a fairly complex routine to follow, particularly for newcomers to machine code. Once you have become knowledgeable in its workings you may want to improve its number handling capabilities to include negative and even floating point numbers. If you enjoy attention to detail you might try to shrink its size so that PATCH is not required — if you succeed, let me know!

As a part of *Database*, SortArray transforms a BASIC program into something quite impressive. As the basis of a machine code filing program it will serve well, but remember that if you are going to write in pure machine code you can arrange your data variables area to suit yourself. The manner in which the BASIC variables are laid out is very useful for implementing variable length strings, but garbage collection then becomes a problem. Having fixed length strings may seem a waste of space but can lead to some very fast searching as well as sorting routines.

The end of this chapter brings us to the end of our sorting routines, but not the end of *Database*. In the next chapter we will add the *Search* facility.

CHAPTER 6
Enhancing *Database*

A search routine

The routines presented in this chapter are all designed to improve the BASIC program introduced in the last chapter, *Database*. The functions they perform, however, may be put to good use in other programs, and the principles involved may be harnessed to your own ends.

The first routine, *Search*, has a self explanatory title. If you give it a target string, it will find a matching one from any field of the BASE$ array and return its number to BASIC. The time savings achieved by using machine code instead of BASIC for this particular purpose are not so dramatic as we have seen with the previous sorting routines, but the method is neat, and it allows the inclusion of so-called wild cards in the target string. These enables us to search for strings without giving a full description.

Search can be used to find a particular entry in the database. If you want to remind yourself of the people who are in your address file whose first name is Paul, then *Search* will find them for you.

A wild card feature enhances *Search*. This allows you to define characters as wild, in which case the routine will match them to any character. Take the example of finding out whose birthdays occur this month; if you have included a date of birth field in your address book database, you may have included it in the form DD-MM-YY, where DD is the date, MM the month and YY the year. Note that you should use two digits for each value, so a birth date of 2nd February 1960 would be entered as 02-02-60. On the other hand, perhaps you might wish to sort the information by age, in which case the form YY-MM-DD would at least allow you to put people in order of the year in which they were born.

Where the wild card feature comes into its own is in finding all the people who were born in, say, June. By giving the target string as '\\-06', where the first two characters are wild, then we can find each entry in the date of birth filed that has '-06' as its third, fourth and fifth characters.

Search is interfaced with *Database* in such a way that the hunt for matching strings begins at the entry after the one currently on display. Therefore, if the 'forename' field is searched for the name 'Paul', the search will begin with the next file. If the last file is reached before a match is found, then *Search* goes

back to the beginning of the aray and continues looking until it has cycled back to the first string that it tested.
If no matching string is found in any of the 'forename' fields of BASE$, the BASIC of *Database* will find a negative value in the result variable RES% and simply re-display the original file. If, however, a match is found, then the number of the file is returned in RES% and *Database* displays that file, making a beep to inform you of its success. To find any further entries that match the target string, you simply call the search function again.

Implementing Search

The incorporation of *Search* into the BASIC of *Database* is achieved by adding the lines of *Program 6.1* to *Program 5.2*. At two points new lines replace parts of the original, while the rest slot into gaps. The bulk of Program 6.1 consists of new data statements, containing the machine code for *Search*; the first data statement alters the size read by the code loading routine, so SortArray is loaded immediately above HIMEM, followed, 232 bytes later, by *Search*.

Program 6.1

```
1070 MAXI%=250:RES%=-1:SCH$=""-"

4125 IF T%=83 OR T%=115 THEN 4500

4500 PRINT #2,"Current search string? ";SCH$;"  (Y/N) ";
4510 INPUT #2,T$
4520 IF T$<>"N" AND T$<>"n" THEN 4550
4530 PRINT #2,"Enter new string ( \ = wild card) ";
4540 LINE INPUT #2,SCH$
4550 INPUT #2,"Which field do you wish to search";WH%
4560 WH%=WH%-1:IF WH%<0 OR WH%>F% THEN 4550
4570 CALL HIMEM+233,EX%+1,@BASE$(0,WH%),@BASE$(IM%,WH%),@SCH$,@RES%
4580 IF RES%>-1 THEN PRINT CHR$(7):EX%=RES%
4590 GOTO 4030

9010 DATA 368

9310 DATA 3E,03,DD,86,04,DD,77,04,768
9320 DATA 30,03,DD,34,05,DD,4E,08,636
9330 DATA DD,46,09,60,69,09,09,DD,740
9340 DATA 5E,06,DD,56,07,19,DD,75,777
9350 DATA 08,DD,74,09,EB,DD,6E,04,924
9360 DATA DD,66,05,A7,ED,52,20,09,855
9370 DATA 01,00,00,DD,5E,06,DD,56,629
9380 DATA 07,C5,DD,6E,02,DD,66,03,863
9390 DATA 1A,4F,46,04,0C,13,23,7E,371
9400 DATA 23,66,6F,EB,7E,23,E5,66,975
9410 DATA 6F,0D,20,03,05,18,0D,05,206
```

```
9420 DATA 28,0A,1A,FE,5C,28,01,BE,653
9430 DATA 13,23,28,ED,D1,13,C1,28,792
9440 DATA 0E,03,DD,6E,08,DD,66,09,688
9450 DATA A7,ED,52,20,B0,CB,F8,DD,1366
9460 DATA 6E,00,DD,66,01,71,23,70,694
9470 DATA C9,00,00,00,00,00,00,00,201
```

By adding the new lines to *Database*, saving and then running the modified program, you will find that the 'S' option of the Examine function becomes active. When it is chosen, you will be asked if you want to enter a new target string, while the previous target string is displayed in brackets. The first time you select *Search* the old target string will be empty; when you enter your chosen string you will be asked which field you wish to search. Having been given a suitable value, the program then calls *Search*, and moments later (microseconds, if you have a small database) the computer will beep and show you the first file in which it found the target string. No beep, no match — you are returned to the original. Finding the next occurrence does not involve you in re-entering the target string.

To include wild card characters in the target string, use the slash symbol (running 'downhill' left to right) to indicate letters you are not interested in.

Although similar in structure to our sorting routines, *Search* uses a different method for detecting the end of its scan through the strings. The code consists of two nested loops (one loop inside another): the outer loop cycles through each file and uses the inner loop to compare the chosen field with the target string. The outer loop ends either when it returns to the first address that it tested or finds a match.

Source code listing 6.1

```
                        100 ;Search.
                        110 ;Finds the first string in an array that matches a given
                        120 ;string, which may include "wild cards".
                        130 ;Searching may begin at any part of an array.
                        140 ;Item at which to begin search, addresses of first, last
                        150 ;and target strings, plus integer variable for result to
                        160 ;be passed by BASIC. On return the result will hold the
                        170 ;the offset of the first element that matches, or a value
                        180 ;one greater than the number of elements if no match.
                        190 ;Position independent - corrupts AF,BC,DE,HL.
7000  3E03              200 SEARCH LD   A,3       ;Add 3 to the "last element"
7002  DD8604            210        ADD  A,(IX+4) ;address.
7005  DD7704            220        LD   (IX+4),A
7008  3003              230        JR   NC,SKIP
700A  DD3405            240        INC  (IX+5)
700D  DD4E08            250 SKIP   LD   C,(IX+8) ;Put number of initial item into BC.
7010  DD4609            260        LD   B,(IX+9)
7013  60                270        LD   H,B      ;Put it in HL and multiply by 3.
7014  69                280        LD   L,C
7015  09                290        ADD  HL,BC
```

```
7016 09          300        ADD  HL,BC
7017 DD5E06      310        LD   E,(IX+6) ;Add the address of the first item.
701A DD5607      320        LD   D,(IX+7)
701D 19          330        ADD  HL,DE
701E DD7508      340        LD   (IX+8),L ;Store "already checked" address in
7021 DD7409      350        LD   (IX+9),H ;IX memory area.
7024 EB          360        EX   DE,HL    ;DE points to item to test.
7025 DD6E04      370 CHKAD  LD   L,(IX+4) ;Fetch address of end of array.
7028 DD6605      380        LD   H,(IX+5)
702B A7          390        AND  A        ;Clear carry.
702C ED52        400        SBC  HL,DE    ;Compare with current address.
702E 2009        410        JR   NZ,INLP
7030 010000      420        LD   BC,0     ;If we have reached the end, restore
7033 DD5E06      430        LD   E,(IX+6) ;BC and pointer to start of array.
7036 DD5607      440        LD   D,(IX+7)
7039 C5          450 INLP   PUSH BC       ;Save counter.
703A DD6E02      460        LD   L,(IX+2) ;Point HL to target string
703D DD6603      470        LD   H,(IX+3) ;descriptor.
7040 1A          480        LD   A,(DE)   ;Put length of test string into C.
7041 4F          490        LD   C,A
7042 46          500        LD   B,(HL)   ;Put length of target string into B.
7043 04          510        INC  B        ;Increment lengths for test purposes.
7044 0C          520        INC  C
7045 13          530        INC  DE       ;Point to addresses of strings.
7046 23          540        INC  HL
7047 7E          550        LD   A,(HL)   ;Get addresses.
7048 23          560        INC  HL
7049 66          570        LD   H,(HL)
704A 6F          580        LD   L,A
704B EB          590        EX   DE,HL    ;DE now points to target string.
704C 7E          600        LD   A,(HL)
704D 23          610        INC  HL
704E E5          620        PUSH HL       ;Save "next descriptor -1"
704F 66          630        LD   H,(HL)
7050 6F          640        LD   L,A      ;HL now points to test string.
     0D          650 TEST   DEC  C        ;If we have not tested whole of test
     2003        660        JR   NZ,STR?  ;string, jump to STR?.
     05          670        DEC  B        ;Set Z if target has been completed.
     180D        680        JR   NXLP?
     05          690 STR?   DEC  B        ;If we have test whole of target
     280A        700        JR   Z,NXLP?  ;string jump to NXLP?.
     1A          710        LD   A,(DE)   ;Test target string for "wild card".
     FE5C        720        CP   #5C
     2801        730        JR   Z,NXELM
     BE          740        CP   (HL)     ;Compare respective characters.
     13          750 NXELM  INC  DE       ;Move pointers.
     23          760        INC  HL
     28ED        770        JR   Z,TEST   ;If match or "wild", jump back.
     D1          780 NXLP?  POP  DE       ;Restore "next descriptor -1" to DE
     13          790        INC  DE       ;Point DE to next descriptor.
     C1          800        POP  BC       ;Retrieve counter.
     280E        810        JR   Z,RESLT  ;If we have found a match, jump.
     03          820        INC  BC       ;Increment counter.
     DD6E08      830        LD   L,(IX+8) ;Load HL with "already tested"
     DD6609      840        LD   H,(IX+9) ;address.
     A7          850        AND  A
     ED52        860        SBC  HL,DE    ;Compare it with current address.
     20B0        870        JR   NZ,CHKAD ;If more to test, jump back.
```

```
CBF8        880        SET  7,B       ;Make counter negative.
DD6E00      890 RESLT  LD   L,(IX+0)  ;Get address of result variable.
DD6601      900        LD   H,(IX+1)
71          910        LD   (HL),C    ;Load it with counter.
23          920        INC  HL
70          930        LD   (HL),B
C9          940        RET
```

Search is documented in *Source Listing 6.1*. It expects a number of values to be passed by BASIC. In order that they should arrive in the IX memory area they are:

@RES%: The address of an integer variable in which *Search* will place the result.

@SCH$: The address of the string descriptor of the target string.

@BASE$(IM%,WH%): The address of the last item in the selected field of the data array.

@BASE$(IM%,0): The address of the first item in the selected field of the array.

EX%+1: The number of the file after the one currently on display: if this is one higher than the total number of items in the field, *Search* will automatically adjust.

The routine, as loaded by *Database*, is located at HIMEM+233. It is fully relocatable, so it can be placed in any safe area of RAM.

Search's first task, when called, is to alter the value in (IX+4) and (IX+5). When BASIC calls *Search*, this parameter contains the address of the descriptor of the last element of the field. By adding 3 to the 16-bit parameter, it will point to an address after the last descriptor of the selected field. This is a useful address to have stored; it can be compared with the address of the descriptor we are about to test to show if we have reached the end of the field. If this is the case, we need to recommence the search at the beginning of the field.

Study how index register operations are used in lines 200-210; often these carry a time penalty, but note that line 240 will be skipped over unless adding 3 to the low byte results in a carry.

BC is used by the outer loop to keep track of the number of the string currently being tested. The value in BC will be returned in the RES% variable as the 'answer' to *Search* — it is *not* used as a loop counter. It would be possible to leave the initial value in the RAM parameter area instead of using a register pair, but the time and size factors of index operations must be considered. Using (INC(IX+d), as we did in line 240, takes three bytes and no less than 23 machine cycles, to say nothing of the further 5 bytes and 12 or even 30 cycles to handle a potential carry into the MSB. POP, INC and PUSH BC total 3 bytes and 26 cycles. Using BC does mean that we have to load the result back into RAM at the end of the program, but this is something that

only needs to be done once, rather than on each pass of the outer loop. The arguments in favour of using the index instructions are founded in keeping the program easy to follow, rather than fast and compact.
Another reason for using BC to hold the file number value rather than one of the other values needed by the routine, is that the initial file number must be fetched from RAM for the purposes of calculating addresses. Having gone to the trouble of fetching the value, we may as well keep it in BC or on the stack.
Lines 270-360 serve two purposes, the most obvious of which is to load DE with the address of the first item we are about to test. In addition, this address is stored in RAM to be used to test for the end of the outer loop. The address could be passed by BASIC, but it is easy for us to calculate it ourselves. The process involves multiplying the number of the first item we wish to test by three, as there are three bytes to each descriptor, and adding the address of the first item in the field. You could argue that we don't multiply the number, but add it to itself twice; the answer is the same.
DE is used as a pointer by the outer loop to keep track of the next string descriptor. It holds an address, rather than a value, and this is checked to see if the outer loop has finished: in other words if all the fields have been searched. I have chosen DE because we will need HL for any address calculations, but we can swap these two pairs with ease (EX DE,HL). BC is normally used as a counter because of the special instructions that make use of it — LDIR and the like, as well as DJNZ. Even when not using these, it helps to work to a pattern — I use B and C individually or as a pair to count with whenever practical.

The outer loop

We enter the loop at CHKAD (CHecK ADdress), line 370. HL is loaded with the address in (IX+4) and (IX+5) that we modified earlier to point past the last entry in the descriptor block in which we are interested. The AND A, SBC HL,DE will result in a carry if we have reached the end — in this event, we reload DE with the address of the first item in the field and reset BC to zero.
The inner loop is prepared for from line 450: BC is saved on the stack. Next you will see that HL is loaded with the address of the target string descriptor — I have used the long-winded index instructions despite what I have said above about size and space, although it should be possible to preserve the address of the target string on the stack. Instinct tells me that savings will be small, if any, and would produce a routine that would be difficult to understand. Another solution to the problem of running out of general purpose register pairs will be demonstrated in the next routine.
We now enter a section already familiar with SortString and SortArray; lines

480-640 load B and C with the respective lengths (+1) of the target and test strings, while DE and HL are loaded with their addresses. In line 600 we save the address of the last byte of the test descriptor on the stack (next descriptor –1) for retrieval at a later stage. Note that HL and DE have now swapped roles, and HL is pointing to the target string, we have used only one EX DE,HL instruction. This is because the next section will need to make use of CP (HL), and there is no CP (DE) instruction. Speed is particularly important in the inner loop which follows because it will be repeated for every character that the routine examines.

The inner loop

Test begins by decrementing C: this is used to keep track of how many untested characters remain in the string. We loaded it with the length of the test string plus one before entering *Test*, so on the first pass we will detect if the test string was empty to begin with. If this is not the case, or on subsequent passes further characters remain to be tested, a jump to STR? is made. However, if the test string was empty or we have been through the loop enough times to have examined the whole of the test string, the routine can decide if a match has been found. In this event the routine decrements B to set the Z flag accordingly and then jumps to the end of the inner loop — if B does not reach zero then the target string is longer than the test string, and by definition a match is not possible; this information is passed in the Z flag. If the routine finds it has characters of the test string remaining to be compared with those of the target, it will have moved on to STR?, where it tests the length of the target string. Remember that it will not have gone through line 670 to get to this point, so B can be decremented to test for zero. If it reaches zero then one of two reasons may be the cause: if this is the first pass of the loop, *Search* has been passed an empty target string: or on subsequent passes we have successfully tested all the characters in the target string against the test string; a match has been found, the Z flag is set and we jump to NXLP?

Lines 710-740 handle the comparison of characters and the wild card feature. On the first pass, the first, or on subsequent passes, further characters of the target string are moved into A. This is compared with &5C, the ASCII code for our wild card character — if they match then a jump is made to NXELM. If you would like Search to treat another character as wild, you can replace line 700 with CP X, (FE X), where X is the ASCII code of your chosen wild card symbol.

Line 740 performs another comparison, this time the character of the target string is compared with the respective character of the test string, setting Z if the characters match.

NXELM increments the address pointers to both strings so as to point to the next characters, and then the conditional jump JR Z,TEST will return us to line 650 if either the target string had contained a wild card or the two characters matched; if this is the case we must continue to cycle through the TEST loop.
You can see that there are three possible exits from the inner loop: the first, at line 670, occurs if all the characters of the test string have been examined: the second, at line 700, happens if all the target characters have been tested: the third is at line 770 — here an exit will only occur if we find two respective characters that do not match. The state of the Z flag indicates why we have exited; if set then we have found a matching string.
We go to the trouble of arranging the Z flag to reflect a match because of lines 780-800. If we jumped to different parts of the routine according to the result of the test, then we would have to have two sets of instructions to restore the stack pointer to its correct value. As written, lines 780-800 are reached in both match and non-match situations.
We POP the 'next descriptor -1' address into DE and then increment it so as to point to the next test string descriptor. The counter is then restored to BC before the decision JR Z,RESLT is taken. If a match has been found, execution transfers to line 890, otherwise the counter is incremented and a test is made to see if a further pass of the outer loop is required. The 'already tested' address is placed in HL and compared with the new descriptor address; if they don't match then another loop is called for so we jump back to CHKAD.

Returning to BASIC

Line 880 is only ever reached by having tested all the strings in the field without finding a match. SET 7,B has the effect of making the counter negative, according to the integer variable convention of Locomotive BASIC. This is how we indicate that no match has been found. At RESLT, the address of the result variable is put into HL and the counter value stored at that address before returning to BASIC.
You may notice that the counter has a maximum capacity of &7FFF (32767), but it is way beyond the memory capacity of the Amstrad to have a string array of that number of elements — the descriptor block alone would occupy 96K bytes!

Two further enhancements

The next routine is designed to speed up a process that, although perfectly possible, would take some time in BASIC. ArrayShift is also a showcase for a

Z80 instruction that we have not used so far — LDIR. One routine allows two functions to be implemented by BASIC: you can move the currently displayed file to the end of the database, thereby allowing a degree of manual sorting: the same routine also enables us to implement a delete function, completely removing the file on display from the records.

Program 6.2 consists of the BASIC required to add move and delete commands to Database: only one line, 9010, replaces a line in the original. Please be aware that Program 6.2 is designed to be used in conjunction with Program 6.1, as it assumes that the Search routine is loaded into memory and places ArrayShift above it. Load the second version of Database, the one to which you have added Program 6.1, and then enter the lines of Program 6.2. Save the new program and then run it. If you have made an error when entering the new DATA statements you will be informed at this point.

Program 6.2

```
4192 PRINT #2,"1 - Alter | 2 - Move to end | 3 - Delete |";
4194 INPUT #2,T%:IF T%<1 OR T%>3 THEN 4192
4196 ON T% GOTO 4200,4300,4400

4300 CALL HIMEM+369,@BASE$(EX%,0),@BASE$(IM%,0),@BASE$(EX%,1),F%+1
4310 GOTO 4260

4400 PRINT #2,"Are you sure? (Y/N)";
4410 INPUT #2,T$
4420 IF T$<>"Y" AND T$<>"y" THEN 4260
4430 CALL HIMEM+369,@BASE$(EX%,0),@BASE$(IM%,0),@BASE$(EX%,1),F%+1
4440 FOR X=0 TO F%
4450  BASE$(IM%,X)=""
4460 NEXT X
4470 IF IM%>0 THEN IM%=IM%-1
4480 GOTO 4260

9010 DATA 440

9480 DATA DD,46,00,78,A7,C8,DD,6E,1109
9490 DATA 02,DD,66,03,DD,5E,04,DD,868
9500 DATA 56,05,A7,ED,52,E5,EB,DD,1262
9510 DATA 5E,06,DD,56,07,A7,ED,52,900
9520 DATA 20,02,E1,C9,E5,FD,E1,EB,1402
9530 DATA D1,D5,54,5D,4E,23,7E,F5,1083
9540 DATA 23,7E,23,C5,FD,E5,C1,ED,1305
9550 DATA B0,C1,D1,2B,77,2B,72,2B,940
9560 DATA 71,D1,19,10,E4,C9,00,00,792
```

If all is well, you will find that using the *Edit* option while examining a file will present you with three choices: altering the file as before, deleting the file or

placing it at the end. This last option is probably more useful than you think: having sorted by a particular field you can sift through the relevant entries placing the ones of interest at the end of the file where they can be printed out as a block. *Delete* means that you can remove a record when it is no longer of interest, reclaiming the memory space that it occupies into the bargain.

Source code listing 6.2

```
                100 ;ArrayShift
                110 ;Moves all the fields of a selected file to the end of the
                120 ;array, shifting all the other files up to fill the gap.
                130 ;Requires the addresses of the the last item in the first
                140 ;field and the selected item in the first and second
                150 ;field, plus the number of fields to be passed by BASIC.
                160 ;Position independent - corrupts AF, BC, DE, HL, IY
DD4600          170         LD   B,(IX+0)   ;Fetch number of fields for use as
78              180         LD   A,B        ;outer loop counter.
A7              190         AND  A          ;Safety check for zero.
C8              200         RET  Z
DD6E02          210         LD   L,(IX+2)   ;Load HL with address of selected
DD6603          220         LD   H,(IX+3)   ;item in second field.
DD5E04          230         LD   E,(IX+4)   ;Load DE with address of last item
DD5605          240         LD   D,(IX+5)   ;in first field.
A7              250         AND  A          ;Makes HL contain offset from end
ED52            260         SBC  HL,DE      ;of one field to begining of next.
E5              270         PUSH HL         ;Save offset on stack.
EB              280         EX   DE,HL      ;Field 1, last item address into HL.
DD5E06          290         LD   E,(IX+6)   ;Field 1, selected item address
DD5607          300         LD   D,(IX+7)   ;into DE.
A7              310         AND  A
ED52            320         SBC  HL,DE      ;Calculate number of descriptor
2002            330         JR   NZ, CONT   ;bytes to move, and if zero,
E1              340         POP  HL         ;restore stack pointer and
C9              350         RET  ;           return to BASIC.
E5              360 CONT    PUSH HL         ;If not zero, place number of bytes
FDE1            370         POP  IY         ;to move into IY.
EB              380         EX   DE,HL      ;Selected item address into HL.
D1              390         POP  DE         ;Get offset into DE.
D5              400 NXFLD   PUSH DE         ;Save DE for begining of loop.
54              410         LD   D,H        ;DE=HL.
5D              420         LD   E,L
4E              430         LD   C,(HL)     ;Fetch selected descriptor bytes,
23              440         INC  HL         ;and point HL to next descriptor.
7E              450         LD   A,(HL)
F5              460         PUSH AF         ;Save descriptor byte 2 on stack.
23              470         INC  HL
7E              480         LD   A,(HL)
23              490         INC  HL
C5              500         PUSH BC         ;Save loop counter and byte 1.
FDE5            510         PUSH IY         ;Number of bytes to be moved into
C1              520         POP  BC         ;BC.
EDB0            530         LDIR ;          Block move.
```

```
C1          540     POP   BC        ;Salvage loop counter and selected
D1          550     POP   DE        ;item descriptor bytes from stack.
2B          560     DEC   HL        ;Point HL to last descriptor byte.
77          570     LD    (HL),A    ;Replace last descriptor with
2B          580     DEC   HL        ;"moved" string descriptor bytes,
72          590     LD    (HL),D    ;and in the process point HL to
2B          600     DEC   HL        ;byte 1 of last descriptor.
71          610     LD    (HL),C
D1          620     POP   DE        ;Get offset from stack.
19          630     ADD   HL,DE     ;HL points to next selected item.
10E4        640     DJNZ  NXFLD     ;If more fields, do loop again.
C9          650     RET
```

The ArrayShift routine

Source Listing 6.2 gives the assembler mnemonics of the 70 bytes of code that make up the routine *ArrayShift*. Four parameters are required from BASIC:

F%+1: the number of fields, counting from 1.

@BASE$(EX%,1): the address of the descriptor of the second field of the currently displayed file.

@BASE$(EX%,0): the address of the descriptor of the first field of the currently displayed file.

ArrayShift has a two-loop structure, although the inner loop consists of only one instruction, LDIR. The outer loop rearranges each field in turn. As before, only the descriptor bytes are moved, not the strings themselves.

All the values required are calculated in the preparation section, lines 170-390. There are also two error traps in order to cope with some of the bad parameters that may be sent by BASIC. Lines 170-200 fetch the number of fields and place them into both B and A. B is to be used as the outer loop counter, while A is tested for zero. If the number of fields sent to the routine happens to be zero, then we return immediately, having done nothing.

Next, the address of the descriptor of the last item in the first field is subtracted from the address of the selected item's descriptor (the one we wish to move) in the first field. This results in a value which is the distance between the end of one field's descriptors and the beginning of the next. We save this value on the stack: it corresponds to the number of bytes that we are going to have to move so that the descriptors above the selected item move down in memory to fill the gap. If this value happens to be zero, it is clear that we are trying to move the last item to the end, which is pointless. It would also cause a nasty crash for reasons I will explain in a moment. So, if the result of the subtraction is zero, no jump is made at line 330. The stack pointer is restored to its correct value and we return to BASIC.

If we do reach CONT, line 360, the result of the subtraction is placed into the IY register. This saves us storing a value in RAM which will be required

inside the loop. The moving of data between the index and general purpose register pairs, however, is not helped by the Z80 instruction set — PUSH and POP are the only sensible way to do this, although they are rather time-consuming at 15 and 14 machine cycles respectively. EX DE,HL places the address of the first item we wish to shift into HL: POP DE retrieves the offset calculated earlier. We are now ready to enter the outer loop.

The outer loop of ArrayShift

The offset will only be used at the end of the loop, so it is put back on the stack to free DE, which is then loaded with HL's contents, so that it points to the descriptor which is to be moved. We now need to collect the three descriptor bytes so that they can be placed at the end of the field after moving the other descriptors. The first is placed in C and HL is incremented; the second is placed in A and saved on the stack. HL is incremented again, and the third descriptor byte to which HL now points is placed in A, where it will be safe for the time being. One more increment of HL and it conveniently points to the next descriptor block. Line 500 saves BC, which contains the loop counter and the first descriptor byte, on the stack. The final two instructions before the inner loop copy the contents of IY into BC.

Let us take stock of the situation. The following information is saved on the stack: the offset to the next field, the second descriptor byte, the first descriptor byte and the loop counter. DE points to the descriptor block which is to be removed to the end; HL points to the next descriptor block, while BC holds the number of bytes that need to be moved.

The inner loop

LDIR, line 530, has the following effect: the byte pointed to by HL is placed at address DE; HL and DE are incremented so that they point to the next bytes, and BC is then decremented. If BC becomes zero then LDIR ceases and control is passed to the next operation. However, if BC does not reach zero then the whole process is repeated over and over again until it does. Therefore, after the LDIR instruction we will have moved all the descriptor bytes of the current field, tha come above the descriptor that we wish to displace, down in memory by three locations; the RAM that contained the original descriptor holds the next, the second has been overwritten by the third, and so on. At the end of the field there are two copies of the last descriptor bytes.

You can see from the description of LDIR that if BC had held zero, the first time it was decremented it would have become FFFF hex — and LDIR would have moved the entire RAM contents of your computer, including all the

BASIC program, the stack and even *ArrayShift*, by three locations. I think you'll agree that this illustrates the efficacy of checking for a request to move a file to the end, when it is already at the end!

Ending the outer loop

Remember that the last three bytes of the descriptor block for the field just processed by the inner loop still hold their original value; we will now proceed to overwrite them with the carefully saved original three bytes of the descriptor of the item we are moving.

The loop counter and first byte are salvaged from the stack (POP BC), and then POP DE places the second byte, saved with the PUSH AF instruction in line 460, into D. At the end of the LDIR instruction, HL was left pointing to the byte after the location in which we want to deposit the displaced descriptor. DEC HL has the effect of pointing HL to where we wish to place the third byte: this was stored in A, so LD (HL),A overwrites the third descriptor byte. Further decrements of HL allow us to place the contents of D and C in their right places.

The whole of one field's descriptors have now been shifted: the item that we wanted to move has been set aside, the subsequent descriptors shifted along to fill the gap, and the displaced descriptor put at the end. We now need to retrieve the offset value from the stack and add it to HL so that it points to the required item in the next field. B is again holding the loop counter. Because the number of fields will be an eight-bit number, DJNZ is all that is required to terminate the outer loop. If another field is to be processed, we jump back to NXFLD (NeXt FieLD); the registers DE, HL and IY hold all the information required for another pass.

ArrayShift is simplified by its use of the LDIR instruction, although this does mean we are forced to use certain registers. Without LDIR we would need to use LD A,(HL); LD (DE),A; INC DE; INC HL; DEC BC; LD A,B; OR C; JR NZ — 52 machine cycles for each pass of the loop. LDIR only takes 21 cycles per pass, and has the additional bonus of not corrupting the A register.

Tying up the loose ends

Database is now almost complete; we have a number of useful routines which allow the information to be manipulated swiftly. One final facility will be of use, not only to *Database* but also to many other BASIC programs. The string sorting method gives lower case characters a higher value than upper case. Forcing all inputted information to upper case is the simplest solution, and to this end we could use the UPPER$ function of Locomotive BASIC, but this would complicate the program.

With a very short machine code routine we can redefine the lower case keys so that they yield upper case letters. This has the advantage that the INPUT statements print out the keys as they appear in the file.

Pulling yourself up by your bootstraps

Redefining the keys need only be done once. It is also time to tidy up the BASIC by removing the code loading routine and the DATA statements, so *Program 6.3* is written as a 'bootstrap' loader for the rest of *Database*. Let us begin with a description of how to create the final version of the *Database* program on cassette — disk users will find all the information they require but need not take any notice of the '!' symbols or mess around getting the programs in order.

Program 6.3

```
100 REM ********************************
101 REM **     Database Bootstrap     **
102 REM ********************************
110 MODE 0
120 LOCATE 3,12
130 PRINT "Loading Database"
140 WHILE 1
150 READ BYTE$:IF BYTE$="END" THEN 180
160 CODE$=CODE$+CHR$(VAL ("&"+BYTE$))
170 WEND
180 CALL PEEK (@CODE$+1)+256*PEEK(@CODE$+2)
190 OPENOUT "DUMMY"
200 MEMORY HIMEM-440
210 LOAD "!CODE",HIMEM+1
220 CLOSEOUT
230 RUN "!DBBASIC"
240 DATA 0E,1A,0C,79,FE,48,C8,CD
250 DATA 30,BB,47,D6,41,D6,1A,30
260 DATA F1,79,CD,27,BB,18,EB,END
```

Enter the *Database Bootstrap* routine, Program 6.3, and check the DATA statements with care — there is no checksum involved so an error here could well lead to a crash. Take a clean cassette and save this new program at the beginning. Now run the BASIC. When you hear the cassette relay click to start the tape, press ESC. You should find that the keyboard behaves as if CAPS LOCK is engaged — try pressing it and you should still get upper case characters. If this is not the case check the DATA statements.

When you have a working version of the bootstrap program, load the last version of *Database*, complete with the *Search*, *Move* and *Delete* modifications, and run it. Once the 'load a database' message appears, ESCape from the program, and enter the following line:

```
START=HIMEM+1:SAVE "CODE",B,START,440
```

Now replace the cassette on which *Bootstrap* was saved, wound past the first program; press play and record and then any key. The machine code will then be saved on tape, immediately following the first BASIC program. Now enter the following lines of BASIC:

```
DELETE 1060
DELETE 8000–9560
```

At this point you may want to list the program and delete the parts that handle lower case letters, such as OR T$="y" in line 1090.

The final step is to SAVE "DBBASIC" on the cassette, following the machine code. Rewind the tape and CATalogue it to check that you have a good recording. The tape now contains the final version of *Database*.

The *Bootstrap* program demonstrates a different method of using machine code, which is suitable for short routines. If the @ prefix is used on a variable name, the address of the variable (or in the case of strings, the descriptor) is returned, rather than the variable itself. This not only works for CALL statements, as used earlier, but also as a function. By creating a string which contains machine code bytes rather than characters, we can store a machine code routine using the BASIC variables system. This saves us having to lower HIMEM, and we can find the location of the code with the @ prefix as shown in line 180.

Two other features are demonstrated by Program 6.3. The OPENOUT "DUMMY" statement, along with the CLOSEOUT (lines 190 and 220), have the effect of permanently reserving a buffer area for cassette operations. This ensures that you don't enter too much data and then find that you run out of memory when you come to save it on tape, and it also avoids a garbage collection occurring every time you wish to use tape.

The second point for cassette users concerns the ! suffix on file names, which disables the prompt messages and avoids the need to "PRESS ANY KEY" when you wish to load a number of consecutive files from tape. This does not appear to be mentioned in my User manual, although it is documented in the Concise BASIC Specification.

CapsLock (Source Listing 6.3) revolves round two firmware routines. &BB30 is the location of KM GET SHIFT. By loading A with a key number (see

Source code listing 6.3

```
                100 ;CapsLock
                110 ;Redefines all lower case keys so that they yield
                120 ;upper case characters.
                130 ;No entry requirements.
                140 ;Position independent - corrupts AF, BC and HL.
0E1A            150 CAPS    LD   C,#1A      ;Load C with first key number -1.
0C              160 NXKEY   INC  C           ;Let C equal next key number.
79              170         LD   A,C         ;Put key number into A.
FE48            180         CP   #48         ;If last key done, return.
C8              190         RET  Z
CD30BB          200         CALL #BB30       ;Get current shifted translation.
47              210         LD   B,A
D641            220         SUB  #41         ;If it is not an upper case letter,
D61A            230         SUB  #1A         ;jump back for next key.
30F1            240         JR   NC,NXKEY
79              250         LD   A,C         ;Put new translation into C.
CD27BB          260         CALL #BB27       ;Alter unshifted translation.
18EB            270         JR   NXKEY       ;Jump back for next key.
```

Appendix III of the User manual) and calling GET SHIFT, the ASCII value of that key is returned in A. &BB27 is KM SET TRANSLATE. If you load A with a key number and B with an ASCII value, and call SET TRANSLATE the character yielded by the key will be altered.

CapsLock itself is simple: it tests keys 26 to 71 (all the letters are contained in this range) to see if they give capital letters in the shifted mode, and if so it redefines the unshifted translation to match. C is used as a form of counter, as B is used by the SET TRANSLATE routine.

Note the use of a conditional return, RET Z. This and the other conditional returns — NZ, C, NC, M, N, PE, PO — are useful, but always remember that the stack pointer must be pointing to the return address; if you have stored anything on the stack you will need to retrieve it before a RET of any kind. For this reason, conditional returns are more likely to be productive in short routines or subroutines.

If you use *CapsLock* in other programs, you may like to know how to reverse the process and restore the keyboard to normal. The easiest way is to CALL &BB00, KM INITIALISE. This sets up the keyboard tables exactly as after switch-on: it corrupts AF, BC, DE and HL. Other firmware routines that may be of use in redefining the keyboard are:

&BB2A — GET TRANSLATE: as GET SHIFT for unshifted keys.
&BB2D — SET SHIFT: as SET TRANSLATE for shifted keys.
&BB33 — SET CONTROL: as SET TRANSLATE when CTRL is pressed.
&BB36 — GET CONTROL: as GET SHIFT when CTRL is pressed.

These, as well as SET TRANSLATE and GET SHIFT, all corrupt AF and

HL. Four more routines with similar responsibilities are:

&BB39 — SET REPEAT: defines whether a key should be allowed to repeat. A must contain the key number; if B is zero the key will not repeat, if &FF it will. Corrupts AF, BC and HL.

&BB3C — GET REPEAT: tells you if a key is set to repeat or not. The answer is contained in the Z flag; if false the key repeats. AF and HL are corrupted.

&BB3F — SET DELAY: sets the delay and repeat parameters. H should hold the delay and L the repeat values in 50ths of a second. Affects all repeating keys, and corrupts AF.

&BB42 — GET DELAY: returns the delay and repeat parameters in H and L respectively. Corrupts AF.

Developing the *Database* program has demonstrated more than just sorting techniques; we have seen how to search and move blocks of data around, as well as how to redefine the keyboard. I hope you will adapt the program to suit your own requirements.

CHAPTER 7

Graphics — a Circle Routine

The purpose of this chapter is two-fold: not only does it present a routine which will be profitable in programs that use graphics, but also the manner in which the routine works demonstrates more useful machine code methods; you will find these helpful in many other applications.
The routine, called *Circle*, adds a function to the Amstrad which is not available as a single command in Locomotive BASIC. Plotting a circle requires some lines of program like this:

```
100 FOR P=-R to +R
110 PLOT XX+P,YY+SQR(R*R-P*P)
120 NEXT P
```

This will, given suitable values for R (the radius), XX and YY (the centre of the circle), plot a passable semi-circle. If you add a second loop to plot negative Y axis co-ordinates, the result will be a circle. To enhance this method, PLOT the first point and then use the DRAW command to achieve a continuous shape. If you try writing a routine to create a circle using the above method, you will find that drawing one which fills the screen will take about 30 seconds, which is painfully slow for most purposes.
How can we speed up this process? Although the Locomotive BASIC resident in the Amstrad is a fast version of the language, if we can by-pass the BASIC interpreter we will gain a great speed advantage. A purpose-written routine in machine code will achieve this. However, the formula which needs to be used to plot a circle involves some fairly complex arithmetic. Consider this:

$$R^*R = X^*X + Y^*Y$$

This relationship is true for all the points around the edge of a circle, centred at X=0, Y=0, of which the radius is R, where X and Y are the horizontal and vertical co-ordinates. We can re-write this as:

$$Y^*Y = R^*R - X^*X$$

or:

Y = SQUARE ROOT (R*R – X*X)

From this formula you can see that to calculate the positions which need to be plotted we must be able to multiply values together, and even more difficult, find the square root of a number. The BASIC interpreter can do this — and so can we; but because the BASIC works to a high degree of accuracy we can save time by limiting our calculations to the scale of numbers we are likely to meet. This, and bypassing the interpreter, are the two main reasons why the *Circle* routine can work about 20 times faster than the equivalent BASIC method.

Now let's prove that it can be done. *Program 7.1* should be entered, along with the code loading routine from Chapter 4. Save the program and run it; provided there are no mistakes in the data statements you will be asked to enter an origin (the centre of the circle), and then the program will draw a series of concentric circles. As written, the code needs to be loaded from address A600; it is not position independent, although you can change it to run at any other location.

Program 7.1

```
100 REM ********************************
101 REM **        Circle Demo           **
102 REM ********************************
110 MEMORY &A66F:GOSUB 8000
120 WHILE 1
130  INPUT "ORIGIN - X";X%
140  INPUT "ORIGIN - Y";Y%
150  CLS
160  FOR R%=20 TO 200 STEP 20
170   CALL &A580,X%,Y%,R%
180  NEXT R%
190 WEND
200 END

9000 REM ********************************
9001 REM **        Circle Data           **
9002 REM ********************************
9010 DATA 240
9020 DATA DD,7E,00,A7,C8,DD,4E,02,1015
9030 DATA DD,46,03,DD,5E,04,DD,56,920
9040 DATA 05,26,00,6F,19,E5,C5,E5,834
```

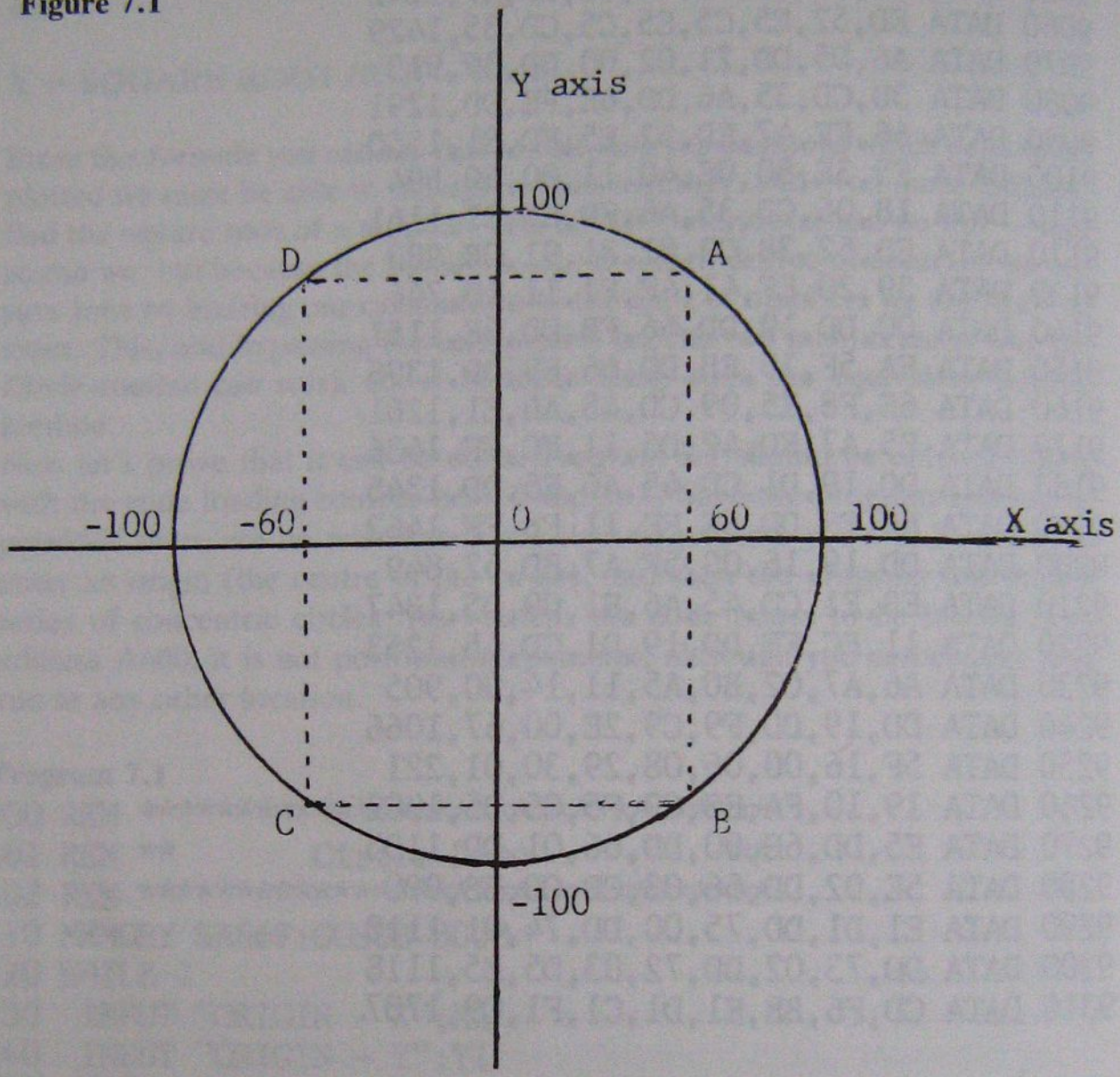
```
9050 DATA C5,EB,E5,C5,16,00,5F,A7,1142
9060 DATA ED,52,E5,C5,E5,C5,CD,35,1429
9070 DATA A6,D5,DD,21,02,00,DD,39,913
9080 DATA 3D,CD,35,A6,DD,6E,FE,DD,1291
9090 DATA 66,FF,A7,ED,52,E5,FD,E1,1550
9100 DATA F5,3E,80,0E,40,11,00,40,594
9110 DATA 18,06,CD,35,A6,FD,E5,E1,1161
9120 DATA ED,52,38,02,81,81,91,CB,983
9130 DATA 39,20,EF,41,4F,F1,11,10,746
9140 DATA 00,DD,19,DD,66,FB,DD,6E,1151
9150 DATA FA,5F,19,EB,DD,66,F9,DD,1398
9160 DATA 6E,F8,E5,09,CD,45,A6,E1,1261
9170 DATA E5,A7,ED,42,D5,11,FC,FF,1436
9180 DATA DD,19,D1,CD,45,A6,E5,DD,1345
9190 DATA 66,FF,DD,6E,FE,11,F8,FF,1462
9200 DATA DD,19,16,00,5F,A7,ED,52,849
9210 DATA EB,E1,CD,45,A6,E1,09,D5,1347
9220 DATA 11,FC,FF,DD,19,D1,CD,45,1253
9230 DATA A6,A7,C2,B0,A5,11,14,00,905
9240 DATA DD,19,DD,F9,C9,2E,00,67,1066
9250 DATA 5F,16,00,06,08,29,30,01,221
9260 DATA 19,10,FA,EB,C9,F5,C5,D5,1382
9270 DATA E5,DD,6E,00,DD,66,01,DD,1105
9280 DATA 5E,02,DD,56,03,CD,C0,BB,990
9290 DATA E1,D1,DD,75,00,DD,74,01,1110
9300 DATA DD,73,02,DD,72,03,D5,E5,1118
9310 DATA CD,F6,BB,E1,D1,C1,F1,C9,1707
```

The method

The circle is plotted in four sections, to keep calculation time to a minimum. The method is best described by an example. Study Figure 7.1. It represents the circle we wish to draw, of radius 100, centered on co-ordinates 0,0. Three-quarters of the circle will not appear on the screen, but we can still work out the theoretical positions that need to be plotted. To plot an arbitrary point, which we will call A, marked in the top left sector, we can apply our formula using an arbitrary value for X:

```
Y = SQUARE ROOT (R*R - X*X)
  = SQUARE ROOT (10000 - 3600)
  = SQUARE ROOT (6400)
  = 80
```

Figure 7.1



Having calculated the relationship between X and Y, it is much simpler to find points B, C and D. If A has the co-ordinates X,Y, then

B=X,−Y: C=X,−Y: and D=−X,Y.

So there are three more points in the circle which we can calculate by simply changing the signs! Therefore

A=60,80: B=60,−80, C=60,−80 and D=−60,80

This saves time, because having calculated the value of Y for a particular value of X, we can plot a point in each of the four quadrants. If the circle is to have an origin somewhere other than 0,0, the co-ordinates of the origin can

Source code listing 7.1

```
                    100 ;Circle
                    110 ;Draws a circle with a maximum radius of 255, centred on
                    120 ;definable user co-ordinates, on the graphics screen.
                    130 ;Uses the current Graphics pen, and works in any mode.
                    140 ;Requires the X and Y co-ords and the radius to be passed
                    150 ;by BASIC. Corrupts all registers.
A580 DD7E00         160         LD   A,(IX+0)   ;Load A with radius,
A583 A7             170         AND  A          ;If radius is 0, return.
A584 C8             180         RET  Z
A585 DD4E02         190         LD   C,(IX+2)   ;BC with Y co-ord,
A588 DD4603         200         LD   B,(IX+3)
A58B DD5E04         210         LD   E,(IX+4)   ;and DE with X co-ord.
A58E DD5605         220         LD   D,(IX+5)
A591 2600           230         LD   H,0
A593 6F             240         LD   L,A
A594 19             250         ADD  HL,DE      ;HL now equals X+radius.
A595 E5             260         PUSH HL         ;Save first two sets of starting
A596 C5             270         PUSH BC         ;co-ords on the stack.
A597 E5             280         PUSH HL
A598 C5             290         PUSH BC
A599 EB             300         EX   DE,HL      ;Load HL with X co-ords.
A59A E5             310         PUSH HL         ;Save X and Y co-ords on stack.
A59B C5             320         PUSH BC
A59C 1600           330         LD   D,0        ;Load DE with radius,
A59E 5F             340         LD   E,A
A59F A7             350         AND  A
A5A0 ED52           360         SBC  HL,DE      ;HL now equals X-radius.
A5A2 E5             370         PUSH HL         ;Save second set of starting
A5A3 C5             380         PUSH BC         ;co-ords on the stack.
A5A4 E5             390         PUSH HL
A5A5 C5             400         PUSH BC
A5A6 CD35A6         410         CALL SQOFA      ;Let DE equal A*A.
A5A9 D5             420         PUSH DE         ;Save radius squared value on stack.
A5AA DD210200       430         LD   IX,2       ;Point IX to data saved on stack.
A5AE DD39           440         ADD  IX,SP
A5B0 3D             450 DRWLP   DEC  A          ;Decrement the X offset.
A5B1 CD35A6         460         CALL SQOFA      ;Get the square of the X offset.
A5B4 DD6EFE         470         LD   L,(IX-2)   ;Load HL with radius squared.
A5B7 DD66FF         480         LD   H,(IX-1)
A5BA A7             490         AND  A
A5BB ED52           500         SBC  HL,DE      ;HL now equals X offset squared
A5BD E5             510         PUSH HL         ;minus radius squared, and moved
A5BE FDE1           520         POP  IY         ;into IY.
A5C0 F5             530         PUSH AF         ;Save loop counter.
A5C1 3E80           540         LD   A,128      ;This section finds the square root
A5C3 0E40           550         LD   C,64       ;of the value in IY. See text.
A5C5 110040         560         LD   DE,#4000
A5C8 1806           570         JR   FRST
A5CA CD35A6         580 SQRLP   CALL SQOFA
A5CD FDE5           590         PUSH IY
A5CF E1             600         POP  HL
A5D0 ED52           610 FRST    SBC  HL,DE      ;NB - carry always cleared by SQOFA.
A5D2 3802           620         JR   C,LOWR
A5D4 81             630         ADD  A,C        ;Double addition saves a jump.
A5D5 81             640         ADD  A,C
A5D6 91             650 LOWR    SUB  C
A5D7 CB39           660         SRL  C
A5D9 20EF           670         JR   NZ,SQRLP
```

```
A5DB  41        680         LD   B,C
A5DC  4F        690         LD   C,A        ;C contains approximate SQR IY.
A5DD  F1        700         POP  AF         ;Retrieve loop counter.
A5DE  111000    710         LD   DE,16      ;Point IX to first segment data.
A5E1  DD19      720         ADD  IX,DE
A5E3  DD66FB    730         LD   H,(IX-5)   ;Load HL with X co-ord.
A5E6  DD6EFA    740         LD   L,(IX-6)
A5E9  5F        750         LD   E,A        ;DE with X offset (D already 0).
A5EA  19        760         ADD  HL,DE
A5EB  EB        770         EX   DE,HL      ;DE now X axis position
A5EC  DD66F9    780         LD   H,(IX-7)   ;Load HL with Y co-ord.
A5EF  DD6EF8    790         LD   L,(IX-8)
A5F2  E5        800         PUSH HL         ;Save Y co-ord on stack.
A5F3  09        810         ADD  HL,BC      ;HL now Y axis position.
A5F4  CD45A6    820         CALL PTSEG      ;Plot segment 1.
A5F7  E1        830         POP  HL         ;Load HL with Y co-ord, but
A5F8  E5        840         PUSH HL         ;keep it on the stack.
A5F9  A7        850         AND  A
A5FA  ED42      860         SBC  HL,BC      ;HL now Y minus C
A5FC  D5        870         PUSH DE         ;Move IX pointer to next
A5FD  11FCFF    880         LD   DE,-4      ;segment's data.
A600  DD19      890         ADD  IX,DE
A602  D1        900         POP  DE
A603  CD45A6    910         CALL PTSEG      ;Plot segment 2.
A606  E5        920         PUSH HL         ;Save Y minus C
A607  DD66FF    930         LD   H,(IX-1)   ;Load HL with X co-ord.
A60A  DD6EFE    940         LD   L,(IX-2)
A60D  11F8FF    950         LD   DE,-8      ;Move IX pointer to segment 3 data.
A610  DD19      960         ADD  IX,DE
A612  1600      970         LD   D,0        ;Load DE with X offset.
A614  5F        980         LD   E,A
A615  A7        990         AND  A
A616  ED52     1000         SBC  HL,DE
A618  EB       1010         EX   DE,HL      ;DE now X minus offset.
A619  E1       1020         POP  HL         ;Retrieve Y minus C.
A61A  CD45A6   1030         CALL PTSEG      ;Plot segment 3.
A61D  E1       1040         POP  HL         ;Get Y co-ord from stack.
A61E  09       1050         ADD  HL,BC      ;HL now Y plus C.
A61F  D5       1060         PUSH DE         ;Point IX to last data block.
A620  11FCFF   1070         LD   DE,-4
A623  DD19     1080         ADD  IX,DE
A625  D1       1090         POP  DE
A626  CD45A6   1100         CALL PTSEG      ;Plot last segment.
A629  A7       1110         AND  A          ;Test X offset for zero.
A62A  C2B0A5   1120         JP   NZ,DRWLP   ;If not, another pass.
A62D  111400   1130         LD   DE,20      ;Restore the stack pointer to
A630  DD19     1140         ADD  IX,DE      ;its original condition.
A632  DDF9     1150         LD   SP,IX
A634  C9       1160         RET  ;Return to BASIC
               1170 ;----   Subroutine to load DE with A*A ----
A635  2E00     1180 SQOFA   LD   L,0        ;Put A in top 8 bits of HL,
A637  67       1190         LD   H,A
A638  5F       1200         LD   E,A        ;and bottom 8 bits of DE.
A639  1600     1210         LD   D,0
A63B  0608     1220         LD   B,8        ;Counter for all powers of 2.
A63D  29       1230 SHIFT   ADD  HL,HL      ;Shifts HL one bit left.
A63E  3001     1240         JR   NC,DONE?   ;If no power in carry, jump.
A640  19       1250         ADD  HL,DE      ;Add orginal value to L.
A641  10FA     1260 DONE?   DJNZ SHIFT      ;Test for further powers.
```

```
A643  EB        1270        EX   DE,HL      ;answer into DE.
A644  C9        1280        RET
                1290 ;---- Subroutine to draw from present position
                1300 ;  in segment to new co-ordinates.------------
A645  F5        1310 PTSEG  PUSH AF         ;Save almost everything!
A646  C5        1320        PUSH BC         ;(The ROM routines corrupt
A647  D5        1330        PUSH DE         ;(AF, BC, DE and HL.)
A648  E5        1340        PUSH HL
A649  DD6E00    1350        LD   L,(IX+0)   ;Get last point drawn to.
A64C  DD6601    1360        LD   H,(IX+1)
A64F  DD5E02    1370        LD   E,(IX+2)
A652  DD5603    1380        LD   D,(IX+3)
A655  CDC0BB    1390        CALL #BBC0      ;Move graphics cursor there.
A658  E1        1400        POP  HL         ;Get new position from stack,
A659  D1        1410        POP  DE
A65A  DD7500    1420        LD   (IX+0),L   ;save in data area,
A65D  DD7401    1430        LD   (IX+1),H
A660  DD7302    1440        LD   (IX+2),E
A663  DD7203    1450        LD   (IX+3),D
A666  D5        1460        PUSH DE         ;and put back on stack.
A667  E5        1470        PUSH HL
A668  CDF6BB    1480        CALL #BBF6      ;Draw line to new position.
A66B  E1        1490        POP  HL         ;Salvage register contents.
A66C  D1        1500        POP  DE
A66D  C1        1510        POP  BC
A66E  F1        1520        POP  AF
A66F  C9        1530        RET
```

have the X and Y values added to or subtracted from them.

To construct a circle we need to find the value of Y for each value of X from the origin to the radius. These co-ordinates, plus those from the other three quadrants, are the points around the edge of the circle.

If we were simply to plot the values, the result would not be a continuous circle — where a change of one unit of X resulted in a larger change in Y, a series of unconnected dots would result. The way round the problem is to draw lines from one point to the next, but this involves us storing the co-ordinates of the last point drawn to of each quadrant — 8 co-ordinates, 16 bytes. *Circle* uses the stack area to store these values.

Circle machine code in detail

Lines 160-250 fetch the integer parameters sent by BASIC (X%,Y% and R%); they are then used to calculate the starting co-ordinates of the first two quadrants which are pushed onto the stack. The co-ordinates of the origin are stacked, and then the third and fourth sets of starting co-ordinates are calculated and pushed. The final piece of data which we wish to store is the value R*R, so that we don't have to calculate it each time we need it: we will use subroutine SQOFA (SQuare OF A), more of which in a moment.

We now have a number of important values stored on the stack; in order to be able to access these easily, IX is pointed to (SP+2) by the instructions LD

IX,2: ADD IX,SP (lines 430-440). We can now get at our data by the use of the index instructions.

Multiplying in binary

SQOFA (lines 1180-1320) is used a number of times. Its purpose is to multiply the value in A by itself and return with the answer in DE, but it can easily be modified to act as a general purpose multiply routine.
First look at this binary multiplication:

Decimal	Binary
137	10001001
*5	00000101
137 (times 1)	10001001
+548 (times 4)	1000100100
=685	1010101101

Study this with care. In order to multiply 137 by five it is first multiplied by one, and to this is added 137 multiplied by four. It is easy to multiply by four in binary, it's just like multiplying by 100 in decimal. We simply move all the digits two spaces to the left and add two zeroes. We know it should be multiplied by four because bit 2 of the second value, five, is set. Similarly, to multiply a number, N, by 137, we split the sum into

N*137 = N*(128 + 8 + 1)

and multiplication by 128 and 8 is simply a matter of shifting digits and adding trailing zeroes in binary. Notice that we can easily see how to break any binary number into these component parts — it's just a case of looking to see which bits are set. We can therefore carry out multiplication in machine code by shifting bits and using addition.

Lines 1180-1210 load H and E with A, and L and D with 0. The B register is to be used as a counter, and it will be decremented by one for each power that is processed. SHIFT, line 1230, adds HL to itself; this has the effect of shifting the contents one bit left as it is in effect, a multiply-by-two operation. The MSB of H moves into the carry. The first pass of the loop will indicate whether A will need to be multiplied by the eighth power of two: if this is the case, DE is added to HL, although this will only perform ADD L,E (but register a carry in bit 0 of H) as D contains zero.
Line 1260, DONE? decrements and tests the loop counter — the loop is performed eight times. Each time the ADD HL,HL instruction is performed,

the value in E will be shifted left one bit — at the end of the loop, the value that we added to E on the first pass (either A or nothing) will have been shifted left eight times; it will occupy the top eight bits of the HL pair and will, in effect, have been multiplied by 256.

During subsequent passes, the lower bits of the original value of H will be shifted into the carry, causing further additions of E to L; these will be shifted left the correct number of times so that, eventually, HL contains A*A. EX DE,HL moves the answer for the convenience of the rest of the program, and then a RET instruction is carried out.

You can modify SQOFA to multiply any eight-bit numbers together. The principle can also be extended to deal with larger numbers, but remember that the answer may well overflow a register pair — a 16-bit value multiplied by an eight-bit value may result in up to a 24-bit number; 16-bit*16-bit=32 bits, and so on.

Calculating the co-ordinates

Let us now look at the main body of the circle routine, which is a loop beginning at line 450. The X co-ordinate offset, held in A, is taken as the loop counter, and begins at one less than the radius. X*X is calculated with the aid of SQOFA, and this value is subtracted from the R*R value, retrieved from the data area. We now have a value of which we wish to find the square root; this is achieved by lines 580-670. I will explain these in detail later on — for now, note that by line 690, BC contains the square root, and therefore the Y axis offset for the current value of the loop counter.

IX is pointed to the first set of co-ordinates, the last point drawn to in the first quadrant. On the first pass of the loop these will be XX+R,YY, where XX and YY are the origin of the circle and R the radius. The new X,Y co-ordinates are calculated by adding the loop counter to XX and the Y offset (stored in BC) to YY: these are placed in DE and HL respectively.

Drawing the circle

A CALL to PTSEG (PrinT SEGment) has the following effect — the graphics cursor is moved to the last point drawn to in each quadrant, the new position stored and then drawn to.

To perform the graphics functions we use ROM routines. The routine to move the graphics cursor, without plotting the actual positon, is GRA MOVE ABSOLUTE, &BBC0. This requires the X and Y co-ordinates to be sent in DE and HL respectively, and corrupts AF, BC, DE, and HL. This explains the need for all the PUSH and POP operations in PTSEG. The other routine, GRA PLOT ABSOLUTE, &BBEA, draws a line from the current position

to the new one supplied in DE and HL (X and Y). The registers affected are the same as for MOVE. I will list some other useful graphics ROM routines later in the chapter.

The rest of the main body of the loop concerns itself with the drawing of the other three segments; finding the negative as well as the positive effects and repeating the process. Assembler lines 830-1100 look, and indeed are, somewhat convoluted in an attempt to keep as many as possible of the values needed close to hand, either in registers or on the stack. BC is used throughout to hold the Y axis offset; this is the value that we went to great pains to calculate. A holds the X axis offset, which is also the loop counter. If you draw connecting lines between each pair of associated PUSH and POPs, it should help you to understand the stack operations. The routine manages to hang on to the Y origin, and only fetches the X origin value once from the data area. Also note the way that the IX pointer is changed for processing each segment so that PTSEG operates on each quadrant in turn; by line 1120, IX is pointing to the same place as at the beginning of the drawing operation.

Line 1110 tests for the end of the routine. If the X offset just plotted, as held in the A register, has reached zero, then we have completed the circle. You may wonder why the decrement operation is at the beginning of the loop but the test at the end. This allows the value of zero to be processed — we cannot quickly test for a value being decremented past zero because DEC A has no effect on the carry flag. This problem could be solved by using SUB 1, but the loop counter (which begins as the radius) needs to be decremented before the first pass of the loop, so putting DEC at the beginning saves an operation.

When the final pass of the loop has been completed, we can return to BASIC; the circle has been drawn. There is one thing remaining to do, however; we have pushed nine two-byte values onto the stack, and they must be removed before a RET will fetch the correct address. We could use nine POPs, or even a DJNZ loop to save program space, but the neatest solution is to calculate where the return address is (LD DE,20: ADD IX,DE) and then reload the stack pointer (LD SP,IX).

Estimating square roots

I have left the explanation of the square root calculation until this point because it is easier to view in isolation, and as a method may well be of use to you in other applications. Let's go through the principle. We want to find the square root of a 16-bit number (0-65536, square roots 0-255), and we begin by guessing it to be 128: this is halfway through the possible answers. The square of 128 is 16384, &4000, which we compare with our target value: if the guess is lower than the target we add half of the original guess (128+64), if higher we

subtract (128–64). We now have a guess of either 192 or 64: we square the appropriate one of these, compare the answer with the target and either add or subtract 32. This process continues, with us repeatedly adding or subtracting descending powers of two; each stage will bring our guess nearer to the right answer.

Binary searching

Now consider the values we are using. We begin with 128; the remaining powers we use are 64, 32, 16, 8, 4, 2 and 1: add these together and the result is 255. Try to reach any number in the range 0-255 by addition and subtraction of the powers and you will find you always get within one of your target: seven operations are the maximum required. We only have an accuracy of 1 in 255, but we can find the approximate square root by this method. You could improve the accuracy by dealing in more places, but for our purposes the errors are acceptable.

The principle of using powers of two to search for the square root of a number is very powerful. It is known as *binary searching*; you are likely to encounter it in sophisticated BASIC programs: there is also a sorting technique, the *Shell-Metzner* sort, which uses it. The crucial point to understand is that by addition and subtraction of descending powers of two we can always get within one place of any value, given that our first guess is a high enough power.

SQRLP — machine code to find a square root

Now let's return to our program. Remember that we want to find the square root of a 16-bit number. The method is helped by the ease with which we can divide a number by two in machine code, just by shifting its bits one place to the right. Lines 540-560 quicken the process by setting up the registers for the first guess; we will always start with 128, the power of which is 16384. Having loaded A with the guessed root, DE with its square (&4000) and C with 64 (half of the first guess), we jump into the middle of SQRLP (SQuare Root LooP).

The first comparison sets the carry flag according to whether an addition or subtraction of C is required to bring the guess closer to the answer. Notice that line 650 avoids the need for two jumps by always subtracting C; so if an addition was required then C must previously have been added not once but twice to cancel the subtraction. If you count bytes and machine cyles you will see that this is a little more efficient, if slightly puzzling.

The value in C is used to determine the end of the loop. Once it has been divided by two (SRL C — 128 becomes 64, etc.), it is tested; if it has become

zero then the loop has finished, otherwise we jump back to SQRLP. The square of the new guess is found by a call to SQOFA: the target is transferred from its storage register, IY, into HL, and a comparison performed by subtraction. The new power of two held by C is added or subtracted as necessary, C divided by two, and the check for another pass made.
At the end of the loop, BC is loaded with A, and we have a sufficiently accurate square root for our purposes. It is possible to improve the accuracy of the approximation: we could jump out of the loop if the comparison resulted in zero. However, this would mean that only some roots would be found accurately; the remaining errors would result in a ragged circle. Another possibility would be to use the remainder of the last comparison to estimate whether to round the result up or down. The number of program bytes that this adds does not, in my opinion, justify the slight improvement to the resulting circle.
One final point worth mentioning about *Circle* is that the result may appear slightly oval; how much it does so will depend on your own particular monitor. The reason for this is that screen pixels are a little larger along the Y axis, while the mathematics of the routine assume them to be as wide as they are high. You could add an adjusting factor to the routine, but remember that what appears perfectly circular on your monitor may not look the same on someone else's.

Graphics firmware routines

The *Circle* routine makes use of the firmware ROM routines. There are a number of other calls that affect the graphics screen, many of which you can utilise. The first time requires the X and Y co-ordinates to be passed in DE and HL respectively. They corrupt AF, BC, DE and HL.

GRA MOVE RELATIVE — &BBC3: a relative version of MOVE ABSOLUTE.
GRA SET ORIGIN — &BBC9: sets a new user origin.
GRA PLOT ABSOLUTE — &BBEA: plots an absolute point.
GRA PLOT RELATIVE — &BBED: a relative plot.
GRA LINE RELATIVE — &BBF9: a relative version of LINE ABSOLUTE.

The next two routines also corrupt the same registers:
GRA CLEAR WINDOW — &BBDB: a CLS of the window display.
GRA INITIALISE — &BBBA: resets all the graphics parameters.

The answers from the next two are returned as X and Y in DE and HL. ASK

CURSOR also corrupts AF:
GRA ASK CURSOR — &BBC6: gets the cursor position.
GRA GET ORIGIN — &BBCC: gets the origin position.

For the next two DE and HL need to contain the X and Y positions or offsets. The result is returned in A; flags, BC, DE and HL are corrupted.
GRA TEST ABSOLUTE — &BBF0: gets the colour of a point.
GRA TEST RELATIVE — &BBF3: relative version of TEST ABSOLUTE.

Both of these require two user co-ordinates, one for each edge to be passed in DE and HL; they corrupt AF, BC, DE and HL.
GRA WIN WIDTH — &BBCF: sets new X axis graphics limits.
GRA WIN HEIGHT — &BBD2: sets new Y axis graphics limits.

In these routines, on return, DE contains the co-ordinates of the left or top edge, HL the right or bottom. AF is corrupted.
GRA GET W WIDTH — &BBD5: gets the X axis limits.
GRA GET W HEIGHT — &BBD8: gets the Y axis limits.

These require A to contain the new colour. Corrupt AF:
GRA SET PEN — &BBDE: sets the graphics pen colour.
GRA SET PAPER — &BBE4: sets the graphics paper colour.

And, finally, these routines have no special requirements:
GRA GET PEN — &BBE1: fetches the current pen colour to A.
GRA GET PAPER — &BBE7: fetches the current paper colour to A.

All the above routines are very robust: they work in whatever screen mode is currently selected and don't mind being sent co-ordinates that are outside the current graphics window. Note that relative co-ordinates may need to be negative; the signed number convention is therefore used, so &FFFF is −1, for example.
Using these routines allows you to perform any of the graphics functions possible from BASIC, except from within your own machine code routines. Shapes of any description can be constructed to suit your particular purpose. For example, drawing squares or triangles would be quite simple to program, as long as you sent the right parameters to your routine.

CHAPTER 8
A Fill Routine

Locomotive BASIC on the 464 does not have a command which will colour in a shape on the screen, other than a rectangular window area. *Fill* is a machine code routine that will do so, and faster, incidentally, than the Fill command on the 664. When you come to study the method used by the routine, you will see that it does have some limitations — filling irregular shapes will require a careful choice of start point, and on occasions more than one *Fill* may be required. These limitations are compensated for by the speed at which the routine works. It is possible to write a shorter routine that has the same effect as *Fill*, using ROM routines to examine the screen one pixel at a time — *Fill* operates on bytes, only resorting to pixel filling when nearing the boundaries of the shape it aims to colour in. To load the routine, enter the code loading routine from Chapter 4 (from tape or disk if possible) and add the program listing 8.1 to it. Save the demo program on tape (we will need it later) and then run it. As long as no data statement errors need correction, the screen will display a diamond shape and you will be asked for a colour in the range 1 to 25. Try 1 to begin with, and you will see the shape quickly filled. Careful examination will reveal that the lower half is filled first, then the upper half.

Program 8.1

```
100 REM ********************************
101 REM **        Fill Demo           **
102 REM ********************************
110 MEMORY &A57F:GOSUB 8000
120 MODE 1:WINDOW £1,1,40,1,1
130 WHILE 1
140 CLS:PLOT 469,199,1
150 DRAW 319,349:DRAW 169,199
160 DRAW 319,49:DRAW 469,199
170 INPUT #1,"Fill colour (1-25)";F%
180 IF F%<1 OR F%>25 THEN 160
190 INK 2,F%
200 CALL &A470,319,199,2
210 IF INKEY$="" THEN 210
```

```
220 WEND
230 END
8000 REM ********************************
8001 REM **   Code Loading Routine   **
8002 REM ********************************
8010 REM   Use lines from Program 4.1
9000 REM ********************************
9001 REM **         Fill Data          **
9002 REM ********************************
9010 DATA 272
9020 DATA CD,CC,BB,44,4D,DD,6E,04,1076
9030 DATA DD,66,05,19,EB,21,7F,02,750
9040 DATA A7,ED,52,D8,DD,6E,02,DD,1256
9050 DATA 66,03,09,44,4D,21,8F,01,436
9060 DATA A7,ED,42,D8,D5,7B,CB,1A,1251
9070 DATA 1F,CB,1A,1F,CB,1A,1F,DD,772
9080 DATA 77,02,2F,C6,50,DD,77,01,787
9090 DATA CD,11,BC,2F,D6,FC,D1,3D,1193
9100 DATA 28,06,CB,2A,CB,1B,18,F7,792
9110 DATA CB,28,CB,19,C5,60,69,CD,1074
9120 DATA 1D,BC,DD,71,05,7E,CB,01,886
9130 DATA 38,03,07,18,F9,CD,2F,BC,779
9140 DATA CD,2C,BC,57,DD,71,03,CB,1064
9150 DATA 09,DD,71,04,DD,7E,00,CD,899
9160 DATA 2C,BC,5F,C1,3E,C7,91,41,991
9170 DATA 4F,04,E5,E5,C5,CD,0C,A5,1120
9180 DATA C1,E1,20,05,CD,26,BC,10,902
9190 DATA F2,E1,41,AF,B8,C8,CD,29,1337
9200 DATA BC,E5,C5,CD,0C,A5,C1,E1,1414
9210 DATA C0,10,F3,C9,DD,4E,05,CD,1161
9220 DATA 6E,A5,C0,CB,09,38,05,CD,945
9230 DATA 5C,A5,30,1B,DD,46,01,AF,799
9240 DATA B8,28,14,E5,CD,20,BC,7E,1024
9250 DATA BA,20,05,73,10,F6,18,06,630
9260 DATA DD,4E,04,CD,5C,A5,E1,DD,1211
9270 DATA 4E,05,CB,01,38,05,CD,65,654
9280 DATA A5,30,17,DD,46,02,AF,B8,888
9290 DATA C8,CD,23,BC,7E,BA,20,04,976
9300 DATA 73,10,F6,C9,DD,4E,03,CD,1085
9310 DATA 65,A5,AF,C9,CD,6E,A5,C0,1314
9320 DATA CB,09,30,F8,C9,CD,6E,A5,1189
9330 DATA C0,CB,01,30,F8,C9,7A,AE,1189
```

```
9340 DATA A1,C0,7B,A1,47,79,2F,A6,1042
9350 DATA 80,77,AF,C9,00,00,00,00,623
```

To show the routine's versatility, break in to the program with ESC and change the screen mode by altering line 120: now restart the program with GOTO 120 and you will see that *Fill* also works in mode 0 and mode 2.

Using *Fill*

To use the program from BASIC you need to send three parameters — the X and Y co-ordinates (user co-ordinates, taking the graphics origin into account) and an ink number. The pixel chosen by the co-ordinates dictates the starting point of the filling, and establishes which is the old ink that is to be overwritten by the new. The form of the CALL is shown in line 200 of the demo program: the parameters need to be in the order X, Y and ink. In the next chapter we will see how to implement *Fill*, as well as *Circle*, as extended BASIC commands.

You may like to experiment with the routine by altering the shape drawn by BASIC in the demonstration program, or by changing the start point. You will soon discover the limitations that the routine acquires as the result of its speed. It can be successful for the sides of a shape, even if they are very irregular, but the top and bottom can pose problems. A careful choice of start point will sometimes help, otherwise further fills will be required.

Before we tackle the method of the machine code routine and details of its coding, it will help if we define the different conventions used to describe screen position, and explore the manner in which the video RAM is used to store information.

Screen co-ordinate systems

Print co-ordinates: this is the most important screen co-ordinate system. You are likely to have used these in conjunction with the LOCATE command in order to move the print position. Character co-ordinates begin at 1,1 — the top left of the screen. Position 2,1 is one character space right, the actual distance depending on the screen mode and therefore the size of the characters. The Y character co-ordinate dictates which character line is printed to — LOCATE 1,25 would move the print position to the bottom of the screen. If a window has been defined, the print co-ordinates begin at the top left corner of the window.

Print co-ordinates are not used by the *Fill* routine, but they do apply to the TXT ROM calls in precisely the same way as they are used by BASIC.

Standard graphics co-ordinates: if you have not moved the graphics origin position with the ORIGIN command, standard graphics co-ordinates and user graphics co-ordinates (as used by the BASIC graphics functions) are identical. The co-ordinates 0, 0 (X,Y) indicate the bottom left corner of the screen. The X co-ordinates run horizontally, with the furthest right position having the value 639. The vertical Y co-ordinates have a maximum value of 399, indicating the top of the screen.

Whatever the screen mode currently in operation, there are always 640 X axis positions. Only in mode 2, however, is there a separate pixel for each position. Mode 1 has 320 pixels per line, so PLOT 0,0 and PLOT 1,0 will produce a dot in the same position. In mode 0, a pixel occupies four co-ordinate positions — PLOT 0,0 and PLOT 3,0 therefore plot the same pixel.

The Y axis contains 200 lines, irrespective of the screen mode, so Y positions 0 and 1 both refer to the bottom line of the screen.

User graphics co-ordinates: as stated above, these only differ from standard co-ordinates if the graphics origin has been moved. Wherever the origin is, it has the co-ordinates 0,0. It is possible to use negative values for user co-ordinates, as these may fall on the screen. For example PLOT −1,−1 will place a dot on the screen one pixel to the left and one line lower than the graphics origin.

Relative graphics co-ordinates: the origin for relative co-ordinates is taken as the last position of the graphics cursor. In all other respects they are the same as user co-ordinates.

Base, or physical, co-ordinates: these are only used by the firmware, and in particular the screen pack. They bear a close resemblance to the actual pixel locations on the screen. The origin, 0,0 is always the bottom left corner of the screen: there are 200 positions on the Y axis, so co-ordinates are in the range 0 to 199. The number of positions on the X axis varies with the screen mode: mode 2 has 640 (0 to 639); mode 1 has 320 (0 to 319); and mode 0 only 160 (0 to 159). To convert standard co-ordinates to base co-ordinates we can divide the Y value by two; the X value is valid for mode 2, and must be divided by two or four for modes 1 or 0 respectively.

The screen memory layout

The Amstrad uses 16K of RAM, addresses C000 to FFFF, to store the information which the video display chip translates into a signal to feed to the monitor or modulator in order to produce a display picture. Almost all of the 16K bytes of RAM store data for the screen display. Let us choose a byte that holds the information for the top left corner of the screen and examine how that data is stored. When in mode 2, the byte holds eight pixels, one in each

bit. The most significant bit (MSB), bit 7, stores the furthest left pixel, that is, the pixel in the very corner of the screen. If the bit is set, that screen pixel is displayed as ink 1 (normally bright yellow, although it can be changed to any colour from the palette of 27). If the bit is zero, then it produces an ink 2 pixel (normally blue).

The second-leftmost bit of the byte, bit 6, holds the data for the second pixel in the top line, (standard co-ordintes 1,399). Subsequent bits relate to the adjacent pixels, until we reach bit 0, which holds the data for standard co-ordinate 7,399. Note that each pixel can only be in of two conditions, which explains why in mode 2 only two colours can be displayed.

The actual colours displayed for the inks are determined by information previously sent to the video chip, which has palette registers in which to hold the information. Changing ink colours is completely unconnected with altering the bytes of screen RAM — it can be performed by a firmware screen routine. Mode 1 stores only four pixels in a byte. Bits 7 and 3 hold the data for the leftmost pixel, bits 6 and 2 hold the second pixel, 5 and 1 the third, and 4 and 0 the rightmost. With two bits available for each pixel, four values can be stored, allowing four inks to be represented. Using the leftmost pixel as an example, bits 3 and 7 are used to store the ink number. The low bit is taken as the most significant — in this case bit 3, so if the byte had a binary value of 00001000, then the leftmost pixel would be displayed as ink 2 (10 binary); bit 3 set (MSB of the ink number of the leftmost pixel and therefore 2) and bit 7 clear. If bit 7 were also set (10001000), the resulting ink number would be 3 — other possibilities are 00000000 (ink 0) and 10000000 (ink 1). In the examples given above, the other pixels all have an ink value of 0, but again in their case, the least significant bit of the byte is always the most significant bit of the ink number. The mode that offers the most inks also has the least number of pixels to a byte (and to a line): mode 0 stores only two pixels per byte. The leftmost pixel's ink number is held in bits 7, 5, 3 and 1; the right-hand pixel of the byte uses bits 6, 4, 2 and 0. As with mode 1, the least significant bit is the most significant in terms of the ink number. If the byte held the information 10000010 binary, the left-hand pixel would be displayed as ink 9.

This may seem a complex arrangement, but the system is designed for the video chip's convenience rather than the programmer's understanding. However, the pixel arrangement lends itself neatly to manipulation by the rotate instructions of the Z80.

The arrangement of the screen memory map

The top 16K of RAM locations hold the pixel information for the screen in an order that is not immediately easy to understand. To increase our problems further, the video display chip can vary the point at which it begins reading

the memory in order to give the effect of a scrolling operation. Before studying the effects of this hardware scroll, let us begin by looking at the memory map in its default state.

The initial video RAM layout

Location C000 is the initial base address of the screen area. The byte of data stored here will give the information for the top left-hand corner of the screen. The bytes C001 to C04F hold the pixel data for the remainder of the top line of the screen. There are always 80 (50 hex) bytes in a screen line, whatever mode is selected — changing screen mode does not alter the order in which the bytes are scanned. From the above discussion of pixel arrangement, you can see that 80 bytes will yield 640, 320 and 160 pixels in modes 2, 1 and 0 respectively.

Bytes following to the first line do not hold the information for the next line down — bytes C050 to C09F hold the data for the ninth screen line, C0A0 to C0EF the data for line 17 and so on. Line 192's data comes from addresses C780 to C7CF; bytes C7D0 to C7FF are unused, providing no data for the screen. The addresses I have just described — C000 to C7FF — form the first 2K block of screen RAM, holding data for the first, ninth and every eighth subsequent line to the bottom of the screen.

The whole of the screen RAM is divided into 2K blocks — while the first block begins with the top line, block 2, addresses C800 to CFD0, begins at the second line. There are eight blocks in all, with the last, F800 to FFFF, beginning at the eighth line and ending with the bottom line of the screen, line 200. These blocks always refer to the same group of lines.

We can look at the arrangement by using a single line of BASIC. Enter the following:

```
MODE 2: FOR X= &C000 TO &FFFF: POKE X,&FF: NEXT X.
```

The screen RAM locations are filled with ink 1 pixels in the order of their addresses. Notice the 'venetian blind' effect as the first, then subsequent, 2K blocks of memory are poked.

The mathematical relationship between pixels on the screen is complex, but can lend itself to quick calculations. If we have the address of a byte somewhere in the middle of the screen, the block of pixels to its right will be one address higher, that to its left one address lower.

In most cases, the screen location above will be stored at an address 2K lower, and that below, 2K higher. On the occasions when adding or subtracting 2K would result in an answer outside the 16K block of screen RAM, the new address can be made valid by setting its two MSBs to one (bringing it into the

range C000-FFFF, thereby restoring the base address), and adding or subtracting a further 80 (50 hex) to move the address one line within the block.

Extra precautions are needed at the edges of the screen, but the above method is quite practical in machine code. Unfortunately, although the relationship between screen addresses and pixel locations remains constant, the hardware scroll feature of the video chip is achieved by adding offsets to the starting addresses — this makes calculation cumbersome.

Changing the base and offset of screen RAM

The base address given above need not be set to C000 — it can be pointed to the beginning of any 16K block of memory. On the Amstrad it is possible to use the block 4000 to 7FFF, although not with a BASIC program. The other two areas contain important operating system software and would be extremely difficult to use.

Although invariably we will find the base address set to C000, it is quite possible that an offset will have been defined which causes the first pixels to come from further along each block. The smallest offset allowed is 2, causing the screen to twitch to the left. An offset of 80 (50 hex) will cause the screen to scroll up by eight pixel lines, and so this is a very quick way of scrolling the screen by one line of characters. Changing the offset will bring the unused bytes at the end of each block into play.

A large offset may mean that we reach the end of a 2K block before we have obtained the required number of bytes. In this case we go back to the beginning of the same block rather than going on to the next. This creates a wrap around effect, causing the first and last bytes of a block to occupy adjacent positions on the screen.

If you are working from machine code, and can therefore guarantee that no offsets will be used, screen address calculations are fairly simple. If you wish to interface a routine with BASIC, you must take a potential offset into account. Doing so is perfectly possible; but unless you are trying to achieve very fast graphic movement, there is no point in re-inventing the wheel — you may as well use the available ROM routines.

Masking — how we isolate individual pixels

The manner in which pixels can be tested and altered without affecting the rest of the byte in which they are stored relies on the use of binary logic operations.

Assuming we are in screen mode 2, the third pixel of a byte is stored in bit 5. How could we isolate the third pixel from all the other data in the byte in

order to test it? Consider what happens if you AND two binary numbers together

```
          10101101
AND       00100000
Result =  00100000
```

If the top number is the screen byte, the effect of the AND operation is to clear all the bits except the one set in the second value — this is called a MASK.

What if the screen byte had been 11011111? Performing an AND between this and the mask 00100000 results in 00000000; so you can see that the mask 00100000 is capable of zeroing all the bits other than bit 5, which retains its original value. If we had used a mask of 00001000, bit 3 would have been isolated, and 10001000 would leave bits 7 and 3 unchanged but clear the rest. It follows that, given the correct mask, we can operate on any pixel by using logic.

The use of masks gives us another bonus: we do not need to keep track of where we are in a byte. Rotating the mask left gives us the mask for the next pixel to the left (RLC 00100000 gives 01000000). In addition, if we try to RLC the bit pattern 10000000, the carry flag will be set, indicating that we have reached the last pixel in the byte.

Masking works just as well in the other screen modes. The mask for pixel 3 in mode 1 is 00100010 — remember there are only four pixels in a byte in this mode — so we need only rotate this twice to the right or once to the left before the carry indicates we have overflowed the end of the byte. A mode 0 mask for pixel 2 would be 01010101, which can only be rotated left once (creating a mask for pixel 1) without causing the carry to be set.

The masking principle is put to good effect by the fill subroutines TSTPX, FLBYR and FLBYL. They work independently of the screen mode and act quickly into the bargain. A study of these routines will give you a good insight into how effects such as smooth pixel movement (rotating a byte and passing the carry over to the next byte) and contrasting cursor printing can be achieved.

The principles behind *Fill*

Let us look at the process that the *Fill* routine follows. The user co-ordinates given are translated into standard and also base co-ordinates, the distances to the sides of the screen calculated, and the address and mask of the first pixel ascertained with the aid of a ROM call. The current ink colour of the chosen pixel is encoded to cover a whole byte, the new ink also encoded into a byte, and masks for the left and right hand pixels created.

The first line is filled by a subroutine (described in a moment). The screen

address is then stepped down a line, and, if this position is still on the screen, the subroutine to fill the line is called again. This process is repeated until either the bottom of the screen is reached or the line filling subroutine returns with the zero flag clear, indicating that the first pixel it tested was not in the old ink. In other words a boundary has been reached.

The original screen address is retrieved. If there are no screen lines above it then we return to BASIC, otherwise the address is pointed one line up the screen, the line filling subroutine called, and this process repeated until the top is reached or a boundary detected.

The routine has now finished. The reason for the top and bottom edges not always being successfully filled is shown up by the above description. Because it is the pixels immediately above and below the starting point that are tested for an old ink value, the routine ceases to fill once a boundary has been detected on that vertical line.

The line fill subroutine — LINFL

The first step is to call a further subroutine — TSTPX. Using the mask and address sent to it, this tests the first pixel to the right; if it finds old ink at the specified pixel, it changes it to new ink and returns with the zero flag set, otherwise the zero flag is cleared.

The line fill routine will study the flag returned to it. If Z is clear, indicating a boundary has been detected, LINFL will return to its calling routine taking the flag with it. If old ink was found, and it has therefore been changed to new ink, the mask is altered so as to isolate the next pixel right and FLBYR called. This fills the current screen byte to its right, with the state of the zero flag indicating a boundary — if one is met, LINFL moves on to filling to the left. Assuming that the first byte is filled to the right without meeting a boundary, the routine now moves on to byte-sized operations, moving the screen address to the right and comparing the byte there with a byte of old ink. If they match, the whole byte is simply replaced with new ink. This loop is continued until either the edge of the screen is reached, or a boundary is detected (i.e. the byte on the screen does not match the old ink byte). In this case, the lefthand mask is fetched and a further call to FLBYR reverts to pixel filling from the left edge of the last byte to the righthand boundary.

*The **Fill** listing*

The *Fill* routine is shown in source code listing 8.1. Note that the three parameters that are expected from BASIC are as follows: (IX+0), the new ink number; (IX+2) and (IX+3), the Y user co-ordinate; (IX+4) and (IX+5), the X user co-ordinate. The remark statements with the listing

Source code listing 8.1

```
                    100 ;FILL: Recolours an area of screen up to pixel boundaries.
                    110 ;Requires first pixel (user co-ords) and new ink to be
                    120 ;passed by BASIC
                    130 ;Corrupts AF, BC, DE, HL.
A470  CDCCBB        140 FILL   CALL  BBCC       ;Get user origin into DE and HL.
A473  44            150        LD    B,H       ;Transfer Y user origin to BC.
A474  4D            160        LD    C,L
A475  DD6E04        170        LD    L,(IX+4)  ;Get X co-ord into HL.
A478  DD6605        180        LD    H,(IX+5)
A47B  19            190        ADD   HL,DE     ;Add origin.
A47C  EB            200        EX    DE,HL
A47D  217F02        210        LD    HL,639    ;Test if X position is off screen.
A480  A7            220        AND   A
A481  ED52          230        SBC   HL,DE
A483  D8            240        RET   C         ;If it is, return.
A484  DD6E02        250        LD    L,(IX+2)  ;Get Y co-ord into HL.
A487  DD6603        260        LD    H,(IX+3)
A48A  09            270        ADD   HL,BC     ;Add origin.
A48B  44            280        LD    B,H
A48C  4D            290        LD    C,L
A48D  218F01        300        LD    HL,399    ;Test if Y position is off screen.
A490  A7            310        AND   A
A491  ED42          320        SBC   HL,BC
A493  D8            330        RET   C         ;If it is, return.
A494  D5            340        PUSH  DE        ;Save X position on stack.
A495  7B            350        LD    A,E       ;Divide X position by 8, putting
A496  CB1A          360        RR    D         ;the result into A.
A498  1F            370        RRA
A499  CB1A          380        RR    D
A49B  1F            390        RRA
A49C  CB1A          400        RR    D
A49E  1F            410        RRA
A49F  DD7702        420        LD    (IX+2),A  ;Save "bytes to left" value.
A4A2  2F            430        CPL   ;         Calculate "bytes to right" value.
A4A3  C650          440        ADD   A,#50
A4A5  DD7701        450        LD    (IX+1),A  ;Save it.
A4A8  CD11BC        460        CALL  #BC11     ;Get screen mode.
A4AB  2F            470        CPL   ;         Transform it for use as counter.
A4AC  D6FC          480        SUB   #FC
A4AE  D1            490        POP   DE        ;Salvage X position.
A4AF  3D            500 BASEL  DEC   A         ;This loop converts the X position
A4B0  2806          510        JR    Z,GETBY   ;to a physical co-ordinate.
A4B2  CB2A          520        SRA   D
A4B4  CB1B          530        RR    E
A4B6  18F7          540        JR    BASEL
A4B8  CB28          550 GETBY  SRA   B         ;Convert Y position to a physical
A4BA  CB19          560        RR    C         ;co-ordinate.
A4BC  C5            570        PUSH  BC        ;Save Y co-ordinate.
A4BD  60            580        LD    H,B
A4BE  69            590        LD    L,C
A4BF  CD1DBC        600        CALL  #BC1D     ;Get screen address and mask.
A4C2  DD7105        610        LD    (IX+5),C  ;Save mask.
A4C5  7E            620        LD    A,(HL)    ;Fetch byte from screen.
A4C6  CB01          630 ODILP  RLC   C         ;This loop moves the desired ink
A4C8  3803          640        JR    C,CONV    ;bits to the left of the byte.
A4CA  07            650        RLCA
A4CB  18F9          660        JR    ODILP
A4CD  CD2FBC        670 CONV   CALL  #BC2F     ;Convert ink information to cover
```

```
A4D0 CD2CBC  680         CALL #BC2C      ;the whole byte.
A4D3 57      690         LD   D,A        ;Put "old ink" byte into D.
A4D4 DD7103  700         LD   (IX+3),C   ;Save "far left" mask.
A4D7 CB09    710         RRC  C          ;Generate "far right" mask.
A4D9 DD7104  720         LD   (IX+4),C   ;Save it.
A4DC DD7E00  730         LD   A,(IX+0)   ;Get "new ink" number.
A4DF CD2CBC  740         CALL #BC2C      ;Convert it to encoded ink byte.
A4E2 5F      750         LD   E,A        ;Put "new ink" byte into E.
A4E3 C1      760         POP  BC         ;Retrieve Y physical co-ord.
A4E4 3EC7    770         LD   A,199      ;Put "lines up" value into C
A4E6 91      780         SUB  C          ;and "lines down" value into B.
A4E7 41      790         LD   B,C
A4E8 4F      800         LD   C,A
A4E9 04      810         INC  B
A4EA E5      820         PUSH HL         ;Save screen address.
A4EB E5      830 DWFIL   PUSH HL
A4EC C5      840         PUSH BC
A4ED CD0CA5  850         CALL LINFL      ;Fill the current screen line.
A4F0 C1      860         POP  BC
A4F1 E1      870         POP  HL
A4F2 2005    880         JR   NZ,UP?     ;If no pixels were filled, jump.
A4F4 CD26BC  890         CALL #BC26      ;Step screen address down a line.
A4F7 10F2    900         DJNZ DWFIL      ;Repeat loop if not off screen.
A4F9 E1      910 UP?     POP  HL         ;Salvage initial screen address.
A4FA 41      920         LD   B,C        ;"Lines up" value into B for use
A4FB AF      930         XOR  A          ;as counter: test for zero, and if
A4FC B8      940         CP   B          ;it is, return.
A4FD C8      950         RET  Z
A4FE CD29BC  960 UPFIL   CALL #BC29      ;Step screen address up a line.
A501 E5      970         PUSH HL
A502 C5      980         PUSH BC
A503 CD0CA5  990         CALL LINFL      ;Fill current line.
A506 C1     1000         POP  BC
A507 E1     1010         POP  HL
A508 C0     1020         RET  NZ         ;If no pixels were filled, return.
A509 10F3   1030         DJNZ UPFIL      ;Repeat loop if not off screen.
A50B C9     1040         RET  ;          Return to BASIC.
            1050 ;Subroutine to fill a screen line.
A50C DD4E05 1060 LINFL   LD   C,(IX+5)   ;Get starting mask into C.
A50F CD6EA5 1070         CALL TSTPX      ;Test and possibly fill pixel.
A512 C0     1080         RET  NZ         ;If pixel was not old ink, return.
A513 CB09   1090         RRC  C          ;If this was the rightmost pixel,
A515 3805   1100         JR   C,LINRT    ;jump ahead.
A517 CD5CA5 1110         CALL FLBYR      ;Fill the pixels to the right, and
A51A 301B   1120         JR   NC,LINLT+1 ;if we don't reach the last, jump.
A51C DD4601 1130 LINRT   LD   B,(IX+1)   ;Get "bytes right" value.
A51F AF     1140         XOR  A          ;If zero, jump ahead.
A520 B8     1150         CP   B
A521 2814   1160         JR   Z,LINLT+1
A523 E5     1170         PUSH HL         ;Save screen address.
A524 CD20BC 1180 RTLOP   CALL #BC20      ;Step screen address a byte right.
A527 7E     1190         LD   A,(HL)     ;Compare the screen contents with a
A528 BA     1200         CP   D          ;byte of old ink, and if they don't
A529 2005   1210         JR   NZ,RTEND   ;match, jump out of the loop.
A52B 73     1220         LD   (HL),E     ;Place new ink byte on screen.
A52C 10F6   1230         DJNZ RTLOP      ;If not off screen, repeat loop.
A52E 1806   1240         JR   LINLT      ;Off the screen - jump ahead.
A530 DD4E04 1250 RTEND   LD   C,(IX+4)   ;Get leftmost mask.
A533 CD5CA5 1260         CALL FLBYR      ;Fill screen byte.
```

```
A536 E1       1270 LINLT  POP  HL          ;Restore screen address.
A537 DD4E05   1280        LD   C,(IX+5)   ;Get original mask.
A53A CB01     1290        RLC  C          ;Move mask one pixel left.
A53C 3805     1300        JR   C,LTDN?    ;If end of byte, jump.
A53E CD65A5   1310        CALL FLBYL      ;Fill byte leftwards.
A541 3017     1320        JR   NC,DONE    ;If not all filled, jump.
A543 DD4602   1330 LTDN?  LD   B,(IX+2)   ;Get "bytes left" as counter.
A546 AF       1340        XOR  A          ;If it is zero, return.
A547 B8       1350        CP   B
A548 C8       1360        RET  Z
A549 CD23BC   1370 LTLOP  CALL #BC23      ;Step screen address left.
A54C 7E       1380        LD   A,(HL)     ;Compare screen byte with old
A54D BA       1390        CP   D          ;ink byte, and if they don't
A54E 2004     1400        JR   NZ,ENDLT   ;match, jump out of loop.
A550 73       1410        LD   (HL),E     ;Put new ink on screen.
A551 10F6     1420        DJNZ LTLOP      ;If not off screen, repeat loop.
A553 C9       1430        RET  ;          Off screen - return.
A554 DD4E03   1440 ENDLT  LD   C,(IX+3)   ;Get rightmost mask.
A557 CD65A5   1450        CALL FLBYL      ;Fill byte leftwards.
A55A AF       1460 DONE   XOR  A          ;Clear zero flag.
A55B C9       1470        RET  ;          Return to calling routine.
              1480 ;Subroutine to fill a byte from mask to right.
A55C CD6EA5   1490 FLBYR  CALL TSTPX      ;Test and possibly set pixel.
A55F C0       1500        RET  NZ         ;If it was not old ink, return.
A560 CB09     1510        RRC  C          ;Rotate mask right.
A562 30F8     1520        JR   NC,FLBYR   ;Repeat loop if not last pixel.
A564 C9       1530        RET
              1540 ;Subroutine to fill a byte from mask to left.
A565 CD6EA5   1550 FLBYL  CALL TSTPX      ;A leftward version of FLBYR.
A568 C0       1560        RET  NZ
A569 CB01     1570        RLC  C
A56B 30F8     1580        JR   NC,FLBYL
A56D C9       1590        RET
              1600 ;Test pixel at (HL), mask c. If it is old ink, change
              1610 ;it to new ink. Zero flag cleared if no match.
A56E 7A       1620 TSTPX  LD   A,D        ;Compare pixel with old ink.
A56F AE       1630        XOR  (HL)
A570 A1       1640        AND  C
A571 C0       1650        RET  NZ         ;Return if no match.
A572 7B       1660        LD   A,E        ;Mask off current pixel of new ink.
A573 A1       1670        AND  C
A574 47       1680        LD   B,A        ;Save in B.
A575 79       1690        LD   A,C        ;Clear current pixel to 0.
A576 2F       1700        CPL
A577 A6       1710        AND  (HL)
A578 80       1720        ADD  A,B        ;Add new ink pixel.
A579 77       1730        LD   (HL),A     ;Put new byte on screen.
A57A AF       1740        XOR  A          ;Clear zero flag.
A57B C9       1750        RET  ;          Return to calling routine.
```

should help you relate the above description of the principle to the component parts of the code. The following sections are of particular interest:
Lines 210-240, 300-330: having calculated the standard co-ordinates, these lines check that the specified point is actually on the screen. If it is not, the routine returns to BASIC having done nothing.
Lines 340-410 divide the X co-ordinate by eight using binary shifting. This

tells the routine how many bytes there are to the left edge of the screen.
Lines 430-440 are a quick way of subtracting the number of bytes to the left from 4F hex, in order to calculate how many bytes there are to the right edge.
Lines 460-540 fetch the screen mode and use it to calculate the X base co-ordinate, while lines 550-560 divide the Y co-ordinate by 2 so it too becomes a base co-ordinate.
Line 600 calls a very useful ROM routine which converts the base co-ordinates to a screen address and mask. The ROM routines used by *Fill*, along with others of interest, will be detailed at the end of this chapter.
Lines 770-810 calculate loop counters from the Y co-ordinate so that reaching the bottom and the top of the screen can be detected.
The main section of *Fill* is contained in lines 820-1040, consisting of two loops, one for the downward and the other for the upward fill.
LINFL, the line filling subroutine described above is contained in lines 1060-1470. Note that the counters used for checking for the edges of the screen are fetched from indexed memory locations, and care is taken to prevent a count of zero, which would cause the loops to be performed 255 times. D and E are used throughout LINFL and its subroutines to hold the old and new ink bytes.
FLBYR and FLBYL (lines 1490-1590) demonstrate the neatness of using circular rotate instructions to alter a mask. There are two possible exits from these routines: where the last pixel in the byte has been treated, or a pixel that is not in old ink is encountered. The state of the flags differs for each exit so that the calling routine can determine what has happened.
The last subroutine, TSTPX (lines 1620-1750), uses both the mask and a complement of it in order to deal with only the desired pixel. Note the neat use of XOR in line 1630 to check for a difference in the bit patterns — the other XOR in line 1740 is simply used to clear the zero flag which indicates that the specified pixel did consist of old ink.

Screen pack firmware routines

Now that the screen layout has been discussed, we can extend the list of screen routines begun in Chapter 3.
BC05 SET OFFSET: alters the offset used by the video chip for the start of each block of memory. HL is sent holding the required offset, and is masked with 7FE to make it legal.
BC08 SET BASE: changes the block of memory used as video RAM. A must pass the high byte of the new block's address, and it is masked with C0 to ensure the RAM begins at a 16K boundary. Both the above routines corrupt AF and HL.
BC0B GET LOCATION: returns the high byte of the base address in A and

the offset in HL.
BC0E SET MODE: changes the screen mode to that sent it in the A register. Corrupts AF, BC, DE, HL.
BC11 GET MODE: returns the current mode number in A.
BC14 CLEAR: sets the whole of the screen to ink 0. Corrupts AF, BC, DE, HL.
BC1D DOT POSITION: When sent the X and Y base co-ordinates in DE and HL respectively, returns the screen address in HL and mask for that pixel in C. B also returns one less than the number of pixels in a byte. Corrupts AF, DE.
BC20 NEXT BYTE: moves a screen address one byte right.
BC23 PREV BYTE: moves a screen address one byte left.
BC26 NEXT LINE: moves a screen address one line down.
BC29 PREV LINE: moves a screen address one line up.
All the above receive and send the address in HL. They do not indicate if you have moved off the screen, which would mean that the returned address would be meaningless. They all corrupt AF.
BC2C INK ENCODE: converts the ink number in A to a whole byte of that ink's bit pattern.
BC2F INK DECODE: takes the leftmost pixel from the byte passed in A and converts it to an ink number.
BC32 SET INK: changes the colour or flashing colours in which an ink is displayed. A should contain the ink number, B the first and C the second colour. Sending B and C at the same value creates a steady colour. Corrupts AF, BC, DE, HL.
BC35 GET INK: if you send an ink number in A, this routine will return the colours of the ink currently being displayed by the video chip via B and C. Corrupts AF, DE, HL.
BC38 SET BORDER: sets the border colour to the values sent in B and C. Corrupts AF, BC, DE, HL.
BC3B GET BORDER: returns the border colour in B and C. Corrupts AF, DE, HL.
BC3E SET FLASHING: sets the time, in TV fields (50th or 60th of a second, depending on country), for which each colour of a flashing pair is displayed. The values are sent in H and L, affecting the first and second colours respectively. Zero is taken to mean 256. Corrupts AF, HL.
The above routines, plus those listed in Chapter 3, give you control over the screen without requiring you to dabble directly with the hardware.

CHAPTER 9

Extending BASIC

The Amstrad CPC computers are equipped with a form of BASIC that is fast and has many excellent features. Although it may be criticised for lacking some commands, it does have one facility that allows virtually any command to be included in its BASIC, and then for it to be used with almost the same ease as if it were an original part of the language. We will employ this feature, called resident system extensions (RSX), so as to implement the *Circle* and *Fill* routines as BASIC commands. We will also create a further keyword, CIRCFL, which will produce a filled circle as long as its centre is within the screen area.

Once an RSX has been loaded an dinitialised, the use of the extra commands is indicated by bar symbol. (|, on the @ key) as a prefix. For example, our circle command will be invoked by using the command |CIRCLE,X,Y,R — whether directly from the keyboard or as part of a program. The parameters are the same as for the direct machine code CALL (the co-ordinates of the centre and the radius). The fill command will take the form |FILL,X,Y,I (I being the ink colour), while the combined command will have the syntax |CIRCFL.X,Y,R,I. Note that a comma is required after the name and before the first parameter. As with other BASIC words, the name need not be typed in upper case letters.

It may seem that we are about to go to a great deal of troube to replace CALL with another word, but RSXs are very simple to install; and CALL is a dangerous command — the wrong address or an incorrect number of parameters can cause a crash, possibly destroying an unsaved program into the bargain.

Having set up the new commands properly, mistyping the name will simply generate an UNKNOWN COMMAND error message. Giving the wrong number of parmeters will cause our own error routine to be invoked. The example shown here is only the tip of the iceberg; whole toolkits of extra commands could be written and incorporated into BASIC.

Setting up the RSX

Program 9.1 gives the necessary data statements for adding our graphics routines to BASIC: it also includes a demonstration of the new CIRCFL command. Enter the listing, not forgetting to include the code loading routine from Chapter 4, and then add a temporary line — 115 STOP. Run the routine to check that the data statements are correct. Once they are, remove the temporary line and save the program.

Program 9.1

```
90 REM NOTE - Circle Demo and Fill Demo
91 REM        MUST be loaded and run
92 REM        before this program!
100 REM ********************************
101 REM **          RSX Demo           **
102 REM ********************************
110 MEMORY &A467:GOSUB 8000
120 CALL &A3F4
130 CLS
140 FOR M=2 TO 0 STEP -1
145 MODE M:Y=0
150  FOR X=225 TO 30 STEP -15
160   Y=Y+1:PLOT -4,-4,Y
170   |CIRCFL,300,200,X,Y
180  NEXT X
190 NEXT M
200 INPUT "Do you wish to save code (Y/N)";A$
210 IF A$="Y" OR A$="y" THEN SAVE "RSX.BIN",B,&A3F0,&280
220 END

9000 REM ********************************
9001 REM **          RSX  Data           **
9002 REM ********************************
9010 DATA 120
9020 DATA 00,00,00,00,21,F0,A3,01,437
9030 DATA FE,A3,CD,D1,BC,C9,09,A4,1393
9040 DATA C3,1A,A4,C3,20,A4,C3,27,1010
9050 DATA A4,46,49,4C,CC,43,49,52,809
9060 DATA 43,4C,C5,43,49,52,43,46,699
9070 DATA CC,00,FE,03,20,20,18,50,629
9080 DATA FE,03,20,1A,C3,80,A5,FE,1057
9090 DATA 04,20,13,DD,23,DD,23,DD,788
9100 DATA E5,CD,80,A5,DD,E1,DD,6E,1504
```

```
9110 DATA FE,DD,75,00,18,32,21,54,783
9120 DATA A4,7E,23,CD,5A,BB,FE,07,1068
9130 DATA 20,F7,06,0A,2B,7C,B5,20,675
9140 DATA FB,10,F9,C9,1E,50,61,72,1038
9150 DATA 61,6D,65,74,65,72,20,65,771
9160 DATA 72,72,6F,72,07,00,00,00,460
```

We must now include the code from the two previous demo programs. This is done by loading the programs and allowing the code loading subroutine to do its work. Load Program 7.1 (Circle Demo) and run it. When you are asked for an origin, ESCape from the program. Then load and run Program 8.1 (Fill Demo), breaking it when asked for a colour.

Now we can reload the RSX DEMO program, and this time we can run it properly. The CIRCFL demo will draw and fill a number of concentric circles in each of the screen modes, ending with a colourful display in mode 0. When the demonstration ends you will be asked if you wish to save the code — I suggest you do this a number of times, and also check your recording so that an error will not cause you to have to repeat the whole procedure. The code saved is all that is needed to implement the commands on subsequent occasions.

Experiment with the new BASIC commands; see what happens when you try CIRCFL with co-ordinates outside the screen area. You could also try deliberately giving the wrong number of parameters to the commands — the error message does not completely halt a BASIC program, but you should not fail to notice it!

The easiest way to incorporate an RSX into an empty computer is to create a bootstrap loading program. *Program 9.2* is an example. If you wish to add the extra commands for use in a larger program, alter the last line (140) to RUN "!TITLE" (where TITLE is the name under which the larger program is to be saved); save the bootstrap program first, then the code (SAVE "RSX",B,&A470,&280), and finally the main program. When you issue the comand RUN "BOOT", the three sections should load consecutively.

Program 9.2

```
100 REM ********************************
101 REM **          RSX Boot            **
102 REM ********************************
110 MEMORY &A3EF
120 LOAD "!RSX.BIN",&A3F0
130 CALL &A3F4
140 NEW
```

RSX tables

The firmware needs a number of conditions to be satisfied before an RSX will work. Firstly, there must be both a command table and a name table stored in memory. The first two locations of the command table must hold the address (LSB first) of the first byte of the name table. There must then follow a three-byte JP command that will cause a jump to the handling routine for each of the new commands.

The name table must contain the names of the commands in the same order in which the JP instructions are stored in the command table. The names are stored as bytes of ASCII codes, with the last character of a name indicated by setting bit 7 high — the last L of FILL would be stored as CC rather than 4C hex. For BASIC to be able to recognise the names they must be in upper case and contain no unusual characters. The end of the name table is indicated by a NUL code, 00 hex or NOP.

The kernel routines require four bytes of RAM for their own purposes, located somewhere between addresses 4000 to BFFF (i.e. not shadowed by ROM). This area can be set up in a small workspace at some convenient location, provided we know the address and there is no danger of the RAM being overwritten. Immediately above a lowered HIMEM is ideal.

The final requirement is for us to "log on" the new RSX. A kernel routine, LOG EXT (BCD1), does this for us. It requires BC to hold the address of the command table, and HL the address of the workspace. The routine corrupts DE. On return, whenever BASIC makes use of another kernel routine, FIND COMMAND (BCD4), it will find our new commands. In the case of a RAM-based RSX, sending a pointer to the desired name in HL will return carry false if no command of that name was found, otherwise HL will contain the address of the relevant JP instruction.

The RSX routine

A study of *source code listing 9.1* will show that our routine fulfills all the above requirements. The tables are detailed in line 170-270, the workspace is at line 1120, while the routine that sets up the registers and calls LOG EXT is located at lines 130-160. This is the only part of the routine that is actually run when we initialise the RSX; the rest serves to implement the commands.

Source code listing 9.1

```
                    100 ;RSX - sets up Circle, Fill and Circf1 as BASIC commands.
                    110 ;No parameters. Corrupts BC, HL.
A3F0  00000000      120 WKSP    DEFB 0,0,0,0      ;4 bytes for kernel use.
A3F4  21F0A3        130 RSX     LD   HL,WKSP      ;Point HL to workspace.
A3F7  01FEA3        140         LD   BC,COM_T     ;Point BC to command table.
A3FA  CDD1BC        150         CALL £BCD1        ;Call KL LOG EXT.
```

```
A3FD  C9          160          RET  ;           End of logging on routine.
A3FE  09A4        170 COM_T    DEFW NAME_T      ;Address of keyword table.
A400  C31AA4      180          JP   FLL         ;Jumps to command handlers.
A403  C320A4      190          JP   CRC
A406  C327A4      200          JP   FCIR
A409  46494C      210 NAME_T   DEFM "FIL"       ;Extended BASIC command names.
A40C  CC          220          DEFB "L"+#80
A40D  43495243    230          DEFM "CIRC"
A411  4CC5        240          DEFB "L","E"+#80
A413  43495243    250          DEFM "CIRC"
A417  46CC        260          DEFB "F","L"+#80
A419  00          270          DEFB 0
A41A  FE03        280 FLL      CP   3           ;If wrong number of parameters
A41C  2020        290          JR   NZ,ERROR    ;jump to ERROR.
A41E  1850        300          JR   #A470       ;Jump to FILL routine.
A420  FE03        310 CRC      CP   3           ;Parameter check.
A422  201A        320          JR   NZ,ERROR
A424  C380A5      330          JP   #A580       ;Jump to CIRCLE routine.
A427  FE04        340 FCIR     CP   4           ;Parameter check.
A429  2013        350          JR   NZ,ERROR
A42B  DD23        360          INC  IX          ;Point IX to "radius" parameter.
A42D  DD23        370          INC  IX
A42F  DDE5        380          PUSH IX          ;Save pointer to parameters.
A431  CD80A5      390          CALL #A580       ;Call CIRCLE routine.
A434  DDE1        400          POP  IX          ;Restore parameter pointer.
A436  DD6EFE      410          LD   L,(IX-2)    ;parameter area.
A439  DD7500      420          LD   (IX+0),L
A43C  1832        430          JR   #A470       ;Jump to FILL routine.
A43E  2154A4      440 ERROR    LD   HL,MSG      ;Point HL to error message.
A441  7E          450 MSLP     LD   A,(HL)      ;Get ASCII code from message.
A442  23          460          INC  HL          ;Point to next character.
A443  CD5ABB      470          CALL #BB5A       ;Output the character.
A446  FE07        480          CP   7           ;If the character was not BELL
A448  20F7        490          JR   NZ,MSLP     ;jump back for next character.
A44A  060A        500          LD   B,10        ;Prepare for delay loop.
A44C  2B          510 DYLP     DEC  HL          ;Inner delay loop decrements
A44D  7C          520          LD   A,H         ;HL until it is zero.
A44E  B5          530          OR   L
A44F  20FB        540          JR   NZ,DYLP
A451  10F9        550          DJNZ DYLP        ;Outer loop repeats 10 times.
A453  C9          560          RET  ;Return to BASIC.
A454  1E          570 MSG      DEFB 30          ;"Home cursor" code.
A455  50617261    580          DEFM "Para"      ;Error message string.
A459  6D657465    590          DEFM "mete"
A45D  72206572    600          DEFM "r er"
A461  726F72      610          DEFM "ror"
A464  07          620          DEFB 7           ;BELL code.
```

When BASIC calls an RSX, it sends, in the A register, the number of parameters it has placed in the IX memory area, and therefore the number of parameters declared to BASIC. The three handling routines begin by checking A: if there are not the correct number of parameters, a jump to the ERROR routine is made; otherwise the CRC and FLL handlers make straightforward jumps to the relevant *Circle* or *Fill* routines.

FCIR adjusts the IX pointer and saves it before calling *Circle*; it then restores IX and loads the ink parameter into the correct location for the *Fill* routine

before jumping to it. Fortunately, *Circle* does not use the IX area for data storage, so the X and Y co-ordinates are left intact for use by *Fill*.

Error handling

The ERROR section demonstrates two classic machine code techniques: a string printing routine and a delay loop. When ERROR is jumped to, the bytes stored at location MSG are sent to the OUTPUT text routine — the first code sets the cursor to the top left of the window, the last causes a short beep to be generated. Note the use of HL as a pointer to the message.

The test for the end of the string uses, in this case, the BELL code (ASCII 7) — however, a number of more general string terminators could be used. If you are unlikely to need the characters 128-255, then you can mark the end of a string by setting bit 7 of the last code; witness the RSX name table. Bit 7 would need to be masked in order to print the last character properly — AND 7F would do the trick. Other possibilities for terminating the string are the NUL code (00 — easily tested for with AND A) or CR (ASCII 13).

The delay consist of an outer loop which repeats 10 hex times: if the length of the delay is not convenient, alter the LD B,10 instruction. The inner loop repeatedly decrements HL until it reaches zero: take note of the OR instruction which is used to test for zero, as decrementing a register pair does not alter the flags. Notice that HL is not set to any starting value, although, after the first pass of the inner loop, it is set to zero so that all further passes will take the longest time possible.

General purpose subroutines

The RSX routine is a good example of how machine code can be exploited to produce programs, as distinct from the routines that we have been writing in this book. It does not perform any major tasks in its own right, but calls other routines to do the work. Some decision-making is involved, and a little parameter manipulation, but any hard work is left to subroutines which can be developed and debugged in isolation from the main program.

The use of subroutines is an important part of machine code writing. Any large program which requires message printing and delay loops would do well to include refined versions of those parts of the error routine. In a moment we will transform them into more general purpose routines which can be included in any major program. But first we must consider what requirements are desirable for general purpose subroutines.

Predictability

Most importantly, they should always do precisely what is expected of them. If they cannot perform the task desired, then they should return some value to the calling program which indicates that they have not succeeded. The TSTPX subroutine in the FILL command is a good example of this — flags are used to indicate whether the pixel tested was found not to be old ink and therefore it was not coloured in with new ink.

As part of this same requirement, the routine should return predictable register values. Apart from returning the resultant values of calculations (witness the screen address ROM routines such as NEXT BYTE), it sometimes may be advantageous to return other information which may, or may not, be used by the calling program — we will use this technique in the message printing subroutine.

If you do not require a register to return a useful value, then it is best to ensure that the original contents are restored. This is normally simple to achieve by using PUSH and POP instructions at the beginning and end of subroutines. The main disadvantage of this technique is that it stops you from using the conditional return instructions — you will need to restore the stack before you can return from the subroutine.

If you do decide to allow a subroutine to corrupt a register (and there are many good reasons for doing so), make sure that you remember which registers are corrupted. Careful documentation will pay dividends here — once you have written a subroutine you will find yourself calling it without much thought to its effects other than the desired one. I vividly remember how I learnt this lesson: having modified a program written some weeks before, I spent a whole day debugging it because sloppy documentation prevented me from appreciating that a particular register had been corrupted.

Flexibility

When you first decide the need for a subroutine, you will probably have a clear view of what is required from it. If you then go on to write a subroutine that meets your specification, it will serve the immediate purpose but there is every possibility that its role could be extended in order also to fulfill other requirements. It is very shortsighted to write a delay routine which produces a fixed delay of five seconds when, with a little extra code, the enhanced version could produce delays varying from milliseconds to minutes.

When written, an examination of the code may reveal that a few more instructions could add extra features. Although these may not have any immediate function, they could be found advantageous at a later stage in the programming.

Building up a library of useful subroutines will save you much time when

tackling large machine code programs. Some of them you may write yourself: others can be gleaned from books and magazine articles. A careful study of your machine's ROM can provide some beneficial material. Another potential source is other programmer's work if, for example, you belong to a computer club, but without proper documentation you will probably find it quicker to start from scratch anyway. Once a useful library has been collected, parts of it can be loaded before you even begin programming, giving you the essence of your own operating system.

Delay

Source code listing 9.2 shows a simple, but effective delay subroutine. The two input parameters, B (about 0.5 second units) and HL (fine adjustment), give a range from a few milliseconds to over two minutes. The fineness of adjustment available means that you can accurately control the running speed of other routines, where, for example, I/O operations are involved. If the longest available delay is insufficient for you, then a *Long Delay* subroutine could be implemented using multiple calls to *Delay*.

Source code listing 9.2

```
                100 ;DELAY - General purpose delay subroutine.
                110 ;Send B as the coarse value, HL as the fine value.
                120 ;Values of B=0, HL=FFFF will give a delay of
                130 ;approximately half a second.
                140 ;Preserves all registers
F5              150 DELAY  PUSH AF
C5              160        PUSH BC
E5              170        PUSH HL
04              180        INC  B
23              190        INC  HL
2B              200 DELLP  DEC  HL
7C              210        LD   A,H
B5              220        OR   L
20FB            230        JR   NZ, DELLP
10F9            240        DJNZ DELLP
E1              250        POP  HL
C1              260        POP  BC
F1              270        POP  AF
C9              280        RET
```

There are only two refinements added to *Delay* which distinguish it from the DYLP section of RSX. All the registers used are preserved on the stack, so *Delay* can be called with impunity. The values sent are incremented before use so that values of 0,0, rather than causing the longest delay, sensibly cause the shortest.

Although I have suggested a long version of *Delay*, a subroutine that I call *Wait* would be a more useful addition to your library. I will not give a listing of this, just a specification — I hope you will take up the challenge and write it yourself. You should make *Wait* capable of generating a delay of up to four minutes, but if a key is pressed during this time it should return: however this should not happen if a key is already pressed when the routine is first called. An indication should be given as to whether the delay was interrupted by a key-press or ran its full length. The most obvious use for *Wait* is the presentation of instruction pages on the screen, accompanied by the message, "Press any key to continue". The specification given makes *Wait* suitable for many applications.

Strings

Source code listing 9.3 contains the essence of the MSLP of RSX in lines 290-400, but it has been expanded to include two *Strings* routines, INSTR and TBSTR: they differ in the manner in which the message string is passed. INSTR expects the message string to be imbedded in program memory immediately after the call; on return, the PC is suitably modified so that execution continues at the instruction after the message. TBSTR expects HL to be pointing to the desired message, allowing us to fetch the string from a table — a string used on more than one occasion need only be included in memory once, and we can also look up the address of a string from another table in order to vary a program's response to different situations.

Source code listing 9.3

```
                    100 ;Strings - two routines to print ASCII strings
                    110 ;including control codes. Code 0 is used to mark
                    120 ;the end of a string. Code 27 will take the
                    130 ;next value as the number of times which to
                    140 ;print the following string of characters.
                    150 ;Both routines leave HL pointing to the byte
                    160 ;after the end of the string, and preserve
                    170 ;all other registers.
                    180 ;INSTR assumes the string is imbedded in
                    190 ;program memory immediately following the call.
                    200 ;TBSTR expects HL to point to the first
                    210 ;character to be printed.
7200  E1            220 INSTR  POP  HL          ;Get previous value of PC.
7201  CD0672        230        CALL TBSTR       ;Print string.
7204  E5            240        PUSH HL          ;Restore return address.
7205  C9            250        RET  ;           Return to after string.
7206  F5            260 TBSTR  PUSH AF          ;Save registers.
7207  C5            270        PUSH BC
7208  D5            280        PUSH DE
7209  7E            290 SRLP   LD   A,(HL)      ;Get character code.
720A  23            300        INC  HL          ;Point to next character.
720B  A7            310        AND  A           ;Check for NUL.
```

```
720C  2004      320        JR   NZ,REPT?   ;If not NUL, jump ahead.
720E  D1        330        POP  DE         ;Restore original register values.
720F  C1        340        POP  BC
7210  F1        350        POP  AF
7211  C9        360        RET
7212  FE1B      370 REPT?  CP   27         ;If the character is "repeat",
7214  2805      380        JR   Z,GETCT    ;jump ahead.
7216  CD5ABB    390        CALL #BB5A      ;Send the character to be printed.
7219  18EE      400        JR   SRLP       ;jump back for another character.
721B  46        410 GETCT  LD   B,(HL)     ;Fetch the repeat value.
721C  23        420        INC  HL         ;Point to next character.
721D  E5        430 RPLP   PUSH HL         ;save address of repeat string.
721E  CD0672    440        CALL TBSTR      ;Recursive call.
7221  EB        450        EX   DE,HL      ;Put end of string into DE.
7222  E1        460        POP  HL         ;Restore HL to start of string.
7223  10F8      470        DJNZ RPLP       ;If more repeats, jump back.
7225  EB        480        EX   DE,HL      ;Point HL to next character.
7226  18E1      490        JR   SRLP       ;Jump back for next character.
                500 ;Demonstration of INSTR.
7228  CD0072    510        CALL INSTR
722B  1B1A      520        DEFB 27,26      ;Repeat 26 times.
722D  54686520  530 MSG    DEFM "The "     ;Message string as ASCII.
7231  636F6465  540        DEFM "code"
7235  20323720  550        DEFM " 27 "
7239  616C6C6F  560        DEFM "allo"
723D  77732079  570        DEFM "ws y"
7241  6F752074  580        DEFM "ou t"
7245  6F207265  590        DEFM "o re"
7249  70656174  600        DEFM "peat"
724D  20737472  610        DEFM " str"
7251  696E6773  620        DEFM "ings"
7255  0000      630        DEFB 0,0        ;End repeat and end string.
7257  C9        640        RET
```

All registers except HL are preserved. On return from either routine, HL will be pointing to the byte after the string just printed. In the case of INSTR this will be the location of the next instruction, and therefore not of use, but after TBSTR it may well be pointing to the next string in the table.

INSTR uses a simple trick to find the address of the string — it POPs the return address from the stack into HL. After calling TBSTR it then PUSHes the new value of HL onto the stack to be used as the return address.

TBSTR has a bonus feature that I decided to include after a study of the character code table in the Amstrad user manual (Chapter 9). There are two codes available if we wish to use all the possible control codes — NUL which does nothing, and ESC, which the manual defines as 'no effect': but the firmware specification goes on to say that it is available to the user. NUL is the best choice as a string terminator — it is easy to test for. How could we exploit ESC?

I decided to use ESC as a 'repeat the following string' code. When encountered, TBSTR jumps to line 410, GETCT (get count), and fetches the next value from the string. It then employs this as a counter to print the next

group of characters up to a NUL, as many times as the counter dictates. This is achieved by a recursive call — TSBSTR actually calls itself! Note how DE is used to store the address after the NUL code, so that when the repeat ceases, we can continue with the rest of the string.

Recursion can be a very powerful tool. It is not normally possible in BASIC because the interpreter will muddle up the variables, using the same ones each time the routine is called. As long as care is taken, machine code can employ recursion to very good effect.

Note that lines 500 onwards are provided for the demonstration of INSTR and the repeat facility. Program 9.3 is a BASIC demo program that will load the subroutines at a low point in memory; you must include the code loading routine from Chapter 4. Having entered, saved and run the program, it should print a message out a number of times. You can add further data statements to the program (suitably modifying line 9010). This causes the program to stop with a data error and you can then enter PRINT SUM to get the correct checksum for the end of each data statement. Using this method you will be able to explore the effects of the control codes and the repeat facility. With a little care you should be able to produce some quite stunning screen displays with just a handful of bytes. The control codes will allow almost anything — you can even draw complex pictures by redefining the character shapes with code 25.

Program 9.3

```
100 REM ********************************
101 REM **         INSTR  Demo          **
102 REM ********************************
110 MEMORY &7257:GOSUB 8000
120 CLS
130 CALL &7228
140 STOP

9000 REM ********************************
9001 REM **         INSTR  Data          **
9002 REM ********************************
9010 DATA 88
9020 DATA E1,CD,06,72,E5,C9,F5,C5,1422
9030 DATA D5,7E,23,A7,20,04,D1,C1,979
9040 DATA F1,C9,FE,1B,28,05,CD,5A,1063
9050 DATA BB,18,EE,46,23,E5,CD,06,994
9060 DATA 72,EB,E1,10,F8,EB,18,E1,1322
9070 DATA CD,00,72,1B,1A,54,68,65,661
9080 DATA 20,63,6F,64,65,20,32,37,580
```

```
9090 DATA 20,61,6C,6C,6F,77,73,20,722
9100 DATA 79,6F,75,20,74,6F,20,72,754
9110 DATA 65,70,65,61,74,20,73,74,790
9120 DATA 72,69,6E,67,73,00,00,C9,748
```

If, having reached the end of this book, you are now itching to write your own machine code, you may be looking for an effective but reasonably straightforward project on which to cut your teeth. Using TBSTR, how many Teletext style screens do you think you could fit into your Amstrad? The screens could be used for many purposes — for example, skipping through like a 'shop window' type of advertising program.

A further refinement would be a program which allowed you to edit and display the results of character code strings, define the order in which they are displayed, and allow different responses from the keyboard to dictate which screen is displayed next. The resulting program could well be used by non-programmers to produced programmed learning educational aids, or information display systems.

Many computer books use literary quotations to punctuate their chapters. If, like us, you find this practice rather peculiar, perhaps you will nevertheless allow us just one with which to end our book; it is too apt for us to resist the temptation.

> These necromantic books are heavenly,
> Lines, circles, scenes, letters and characters:
> Ay, these are those that Faustus most desires.
> Oh, what a world of profit and delight,
> Of power, of honour, of omnipotence,
> Is promised to the studious artizan!
>
> Christopher Marlowe
> Doctor Faustus 1.1.

APPENDIX 1

Z80 Instruction Set

The mnemonics can be understood by reference to Chapter 1. The codes are given in hexadecimal, followed by the number of machine cycles the instruction requires for execution — in the case of two figures being given, the instruction is conditional and the first figure applies to the condition being met.

ADC A,A	8F	4	ADD A,L	85	7
ADC A,B	88	4	ADD A,data	C6 dd	7
ADC A,C	89	4	ADD A,(HL)	86	7
ADC A,D	8A	4	ADD A,(IX+x)	DD 86 xx	19
ADC A,E	8B	4	ADD A,(IY+x)	FD 86 xx	19
ADC A,H	8C	4	ADD HL,BC	09	11
ADC A,L	8D	4	ADD HL,DE	19	11
ADC A,data	CE dd	7	ADD HL,HL	29	11
ADC A,(HL)	8E	7	ADD HL,SP	39	11
ADC A,(IX+x)	DD 8E xx	19	ADD IX,BC	DD 09	15
ADC A,(IY+x)	FD 8E xx	19	ADD IX,DE	DD 19	15
ADC HL,BC	ED 4A	15	ADD IX,IX	DD 29	15
ADC HL,DE	ED 5A	15	ADD IX,SP	DD 39	15
ADC HL,HL	ED 6A	15	ADD IY,BC	FD 09	15
ADC HL,SP	ED 7A	15	ADD IY,DE	FD 19	15
ADD A,A	87	4	ADD IY,IY	FD 29	15
ADD A,B	80	4	ADD IY,SP	FD 39	15
ADD A,C	81	4	AND A	A7	4
ADD A,D	82	4	AND B	A0	4
ADD A,E	83	4	AND C	A1	4
ADD A,H	84	7	AND D	A2	4

AND E	A3	4
AND H	A4	4
AND L	A5	4
AND data	E6 dd	7
AND (HL)	A6	7
AND (IX+x)	DD A6 xx	19
AND (IY+x)	FD A6 xx	19
BIT 0,A	CB 47	8
BIT 0,B	CB 40	8
BIT 0,C	CB 41	8
BIT 0,D	CB 42	8
BIT 0,E	CB 43	8
BIT 0,H	CB 44	8
BIT 0,L	CB 45	8
BIT 0,(HL)	CB 46	12
BIT 0,(IX+x)	DD CB xx 46	20
BIT 0,(IY+x)	FD CB xx 46	20

For other bit numbers, add the following to the 2nd byte of the instruction (4th for the index codes):

BIT 1 +8	BIT 5 +28
BIT 2 +10	BIT 6 +30
BIT 3 +18	BIT 7 +38
BIT 4 +20	

CALL addr	CD al ah	17
CALL C,addr	DC al ah	17/10
CALL M,addr	FC al ah	17/10
CALL NC,addr	D4 al ah	17/10
CALL NZ,addr	C4 al ah	17/10
CALL P,addr	F4 al ah	17/10
CALL PE,addr	EC al ah	17/10
CALL PO.addr	E4 al ah	17/10
CALL Z,addr	CC al ah	17/10
CCF	3F	4
CP A	BF	4
CP B	B8	4
CP C	B9	4
CP D	BA	4
CP E	BB	4
CP H	BC	4
CP L	BD	4
CP data	FE dd	7
CP (HL)	BE	7
CP (IX+x)	DD BE xx	19
CP (IY+x)	FD BE xx	19
CPD	ED A9	16
CPDR	ED B9	21/16
CPI	ED A1	16
CPIR	ED B1	21/16
CPL	2F	4
DAA	27	4
DEC A	3D	4
DEC B	05	4
DEC C	0D	4
DEC D	15	4
DEC E	1D	4
DEC H	25	4
DEC L	2D	4
DEC (HL)	35	11
DEC (IX+x)	DD 35 xx	23
DEC (IY+x)	FD 35 xx	23
DEC BC	0B	6
DEC DE	1B	6
DEC HL	2B	6

DEC IX	DD 2B	10	INC (IX+x)	DD 34 xx	23
DEC IY	FD 2B	10	INC (IY+x)	FD 34 xx	23
DEC SP	3B	6	INC BC	03	6
DI	F3	4	INC DE	13	6
DJNZ,dis	10 ds	13/8	INC HL	23	6
EI	FB	4	INC IX	DD 23	10
EX DE,HL	EB	4	INC IY	FD 23	10
EX AF,AF'	08	4	INC SP	33	6
EX (SP),HL	E3	19	IND	ED AA	16
EX (SP),IX	DD E3	23	INDR	ED BA	21/16
EX)SP),IY	FD E3	23	INI	ED A2	16
EXX	D9	4	INDIR	ED B2	21/16
HALT	76	4	JP addr	C3 al ah	10
IM 0	ED 46	8	JP C,addr	DA al ah	10
IM 1	ED 56	8	JP M,addr	FA al ah	10
IM 2	ED 5E	8	JP NC,addr	D2 al ah	10
IN A,(C)	ED 78	12	JP NZ,addr	C2 al ah	10
IN B,(C)	ED 40	12	JP P,addr	F2 al ah	10
IN C,(C)	ED 48	12	JP PE,addr	EA al ah	10
IN D,(C)	ED 50	12	JP PO,addr	E2 al ah	10
IN E,(C)	ED 58	12	JP Z,addr	CA al ah	10
IN H,(C)	ED 60	12	JP (HL)	E9	4
IN L,(C)	ED 68	12	JP (IX)	DD E9	8
IN A,(port)	DB pt	11	JP (IY)	FD E9	8
INC A	3C	4	JR dis	18 ds	12
INC B	04	4	JR C,dis	38 ds	12/7
INC C	0C	4	JR NC,dis	30 ds	12/7
INC D	14	4	JR NZ,dis	20 ds	12/7
INC E	1C	4	JR Z,dis	28 ds	12/7
INC H	24	4	LD (addr),A	32 al ah	13
INC L	2C	4	LD (addr),BC	ED 43 al ah	20
INC (HL)	34	11	LD (addr),DE	ED 53 al ah	20

LD (addr),HL	ED 63 al ah	20
LD (addr),HL	22 al ah	16
LD (addr),IX	DD 22 al ah	20
LD (addr),IY	FD 22 al ah	20
LD (addr),SP	ED 73 al ah	20
LD (BC),A	02	7
LD (DE),A	12	7
LD (HL),A	77	7
LD (HL),B	70	7
LD (HL),C	71	7
LD (HL),D	72	7
LD (HL),E	73	7
LD (HL),H	74	7
LD (HL),L	75	7
LD (HL),data	36 dd	7
LD (IX+x),A	DD 77 xx	19
LD (IX+x),B	DD 70 xx	19
LD (IX+x),C	DD 71 xx	19
LD (IX+x),D	DD 72 xx	19
LD (IX+x),E	DD 73 xx	19
LD (IX+x),H	DD 74 xx	19
LD (IX+x),L	DD 75 xx	19
LD (IX+x),dt	DD 36 xx dd	19
LD (IY+x),A	FD 77 xx	19
LD (IY+x),B	FD 70 xx	19
LD (IY+x),C	FD 71 xx	19
LD (IY+x),D	FD 72 xx	19
LD (IY+x),E	FD 73 xx	19
LD (IY+x),H	FD 74 xx	19
LD (IY+x),L	FD 75 xx	19
LD (IY+x),dt	FD 36 xx dd	19
LD A,(addr)	3A al ah	13
LD A,(BC)	0A	7
LD A,(DE)	1A	7
LD A,(HL)	7E	7
LD,A,A	7F	4
LD A,B	78	4
LD A,C	79	4
LD A,D	7A	4
LD A,E	7B	4
LD A,H	7C	4
LD A,I	ED 57	9
LD A,L	7D	4
LD A,R	ED 5F	9
LD A,data	3E dd	7
LD A,(IX+x)	DD 7E xx	19
LD A,(IY+x)	FD 7E	19
LD B,A	47	4
LD B,B	40	4
LD B,C	41	4
LD B,D	42	4
LD B,E	43	4
LD B,H	44	4
LD B,L	45	4
LD B,data	06 dd	7
LD B,(HL)	46	7
LD B,(IX+x)	DD 46 xx	19
LD B,(IY+x)	FD 46 xx	19
LD BC,data	01 dl dh	10
LD BC,(addr)	ED 4B al ah	20
LD C,A	4F	4
LD C,B	48	4
LD C,C	49	4
LD C,D	4A	4

LD C,E	4B	4	LD H,B	60	4
LD C,H	4C	4	LD H,C	61	4
LD C,L	4D	4	LD H,D	62	4
LD C,data	0E dd	7	LD H,E	63	4
LD C,(HL)	4E	7	LD H,H	64	4
LD C,(IX+x)	DD 4E xx	19	LD H,L	65	4
LD C,(IY+x)	FD 4E xx	19	LD H,data	26 dd	7
LD D,A	57	4	LD H,(HL)	66	7
LD D,B	50	4	LD H,(IX+x)	DD 66 xx	19
LD D,C	51	4	LD H,(IY+x)	FD 66 xx	19
LD D,D	52	4	LD HL,data	21 dl dh	10
LD D,E	53	4	LD HL,(addr)	ED 6B al ah	21
LD D,H	54	4	LD HL,(addr)	2A al ah	16
LD D,L	55	4	LD I,A	ED 47	9
LD D,data	16 dd	4	LD IX,data	DD 21 dl dh	14
LD D,(HL)	56	7	LD IX,(addr)	DD 2A al ah	20
LD D,(IX+x)	DD 56 xx	19	LD IY,data	FD 21 dl dh	14
LD D,(IY+x)	FD 56 xx	19	ID IY,(addr)	FD 2A al ah	20
LD DE,data	11 dl dh	10	LD L,A	6F	4
LD DE,(addr)	ED 5B al ah	20	LD L,B	68	4
LD E,A	5F	4	LD L,C	69	4
LD E,B	58	4	LD L,D	6A	4
LD E,C	59	4	LD L,E	6B	4
LD E,D	5A	4	LD L,H	6C	4
LD E,E	5B	4	LD L,L	6D	4
LD E,H	5C	4	LD L,data	2E dd	7
LD E,L	5D	4	LD L,(HL)	6E	7
LD E,data	1E dd	7	LD L,(IX+x)	DD 6E xx	19
LD E,(HL)	5E	7	LD L,(IY+x)	FD 6E xx	19
LD E,(IX+x)	DD 5E xx	19	LD R,A	ED 4F	9
LD E,(IY+x)	FD 5E xx	19	LD SP,data	31 dl dh	10
LD H,A	67	4	LD SP,(addr)	ED 7B al ah	20

LD SP,HL	F9	6
LD SP,IX	DD F9	10
LD SP,IY	FD F9	10
LDD	ED A8	16
LDDR	ED B8	21/16
LDI	ED A0	16
LDIR	ED B0	21/16
NEG	ED 44	8
NOP	00	4
OR A	B7	4
OR B	B0	4
OR C	B1	4
OR D	B2	4
OR E	B3	4
OR H	B4	4
OR L	B5	4
OR data	F6 dd	7
OR (HL)	B6	7
OR (IX+x)	DD B6 xx	19
OR (IY+x)	FD B6 xx	19
OTDR	ED BB	21/16
OTIR	ED B3	21/16
OUT (C),A	ED 79	12
OUT (C),B	ED 41	12
OUT (C),C	ED 49	12
OUT (C),D	ED 51	12
OUT (C),E	ED 59	12
OUT (C),H	ED 61	12
OUT (C),L	ED 69	12
OUT (port),A	D3 pt	11
OUTD	ED AB	16
OUTI	ED A3	16
POP AF	F1	10
POP BC	C1	10
POP DE	D1	10
POP HL	E1	10
POP IX	DD E1	14
POP IY	FD E1	14
PUSH AF	F5	11
PUSH BC	C5	11
PUSH DE	D5	11
PUSH HL	E5	11
PUSH IX	DD E5	15
PUSH IY	FD E5	15
RES 0,A	CB 87	8
RES 0,B	CB 80	8
RES 0,C	CB 81	8
RES 0,D	CB 82	8
RES 0,E	CB 83	8
RES 0,H	CB 84	8
RES 0,L	CB 85	8
RES 0,(HL)	CB 86	15
RES 0,(IX+x)	DD CB xx 86	23
RES 0,(IY+x)	FD CB xx 86	23

See BIT for information about how to calculate other bits.

RET	C9	10
RET C	D8	11/5
RET M	F8	11/5
RET NC	D0	11/5
RET NZ	C0	11/5
RET P	F0	11/5
RET PE	E8	11/5
RET PO	E0	11/5

RET Z	C8	11/5	RR L	CB 1D	8
RETI	ED 4D	14	RR (HL)	CB 1E	15
RETN	ED 45	14	RR (IX+x)	DD CB xx 1E	23
RL A	CB 17	8	RR (IY+x)	FD CB xx 1E	23
RL B	CB 10	8	RRA	1F	4
RL C	CB 11	8	RRC A	CB 0F	8
RL D	CB 12	8	RRC B	CB 08	8
RL E	CB 13	8	RRC C	CB 09	8
RL H	CB 14	8	RRC D	CB 0A	8
RL L	CB 15	8	RRC E	CB 0B	8
RL (HL)	CB 16	15	RRC H	CB 0C	8
RL (IX+x)	DD CB xx 16	23	RRC L	CB 0D	8
RL (IY+x)	FD CB xx 16	23	RRC (HL)	CB 0E	8
RLA	17	4	RRC (IX+x)	DD CB xx 0E	23
RLC A	CB 07	8	RRC (IY+x)	FD CB xx 0E	23
RLC B	CB 00	8	RRCA	0F	4
RLC C	CB 01	8	RRD	ED 67	18
RLC D	CB 02	8	RST 0 (00)	C7	11
RLC E	CB 03	8	RST 1 (08)	CF	11
RLC H	CB 04	8	RST 2 (10)	D7	11
RLC L	CB 05	8	RST 3 (18)	DF	11
RLC (HL)	CB 06	15	RST 4 (20)	E7	11
RLC (IX+x)	DD CB xx 06	23	RST 5 (28)	EF	11
RLC (IY+x)	FD CB xx 06	23	RST 6 (30)	F7	11
RLCA	07	4	RST 7 (38)	FF	11
RLD	ED 6F	18	SBC A,A	9F	4
RR A	CB 1F	8	SBC A,B	98	4
RR B	CB 18	8	SBC A,C	99	4
RR C	CB 19	8	SBC A,D	9A	4
RR D	CB 1A	8	SBC A,E	9B	4
RR E	CB 1B	8	SBC A,H	9C	4
RR H	CB 1C	8	SBC A,L	9D	4

SBC A,data	DE dd	7
SBC A,(HL)	9E	7
SBC A,(IX+x)	DD 9E xx	19
SBC A,(IY+x)	FD 9E xx	19
SCF	37	4
SET 0,A	CB C7	8
SET 0,B	CB C0	8
SET 0,C	CB C1	8
SET 0,D	CB C2	8
SET 0,E	CB C3	8
SET 0,H	CB C4	8
SET 0,L	CB C5	8
SET 0,(HL)	CB C6	15
SET 0,(IX+x)	DD CB xx C6	23
SET 0,(IY+x)	DD CB xx C6	23

See BIT for information about how to calculate other bits.

SLA A	CB 27	8
SLA B	CB 20	8
SLA C	CB 21	8
SLA D	CB 22	8
SLA E	CB 23	8
SLA H	CB 24	8
SLA L	CB 25	8
SLA (HL)	CB 26	15
SLA (IX+x)	DD CB xx 26	23
SLA (IY+x)	FD CB xx 26	23
SRA A	CB 2F	8
SRA B	CB 28	8
SRA C	CB 29	8
SRA D	CB 2A	8
SRA E	CB 2B	8
SRA H	CB 2C	8
SRA L	CB 2D	8
SRA (HL)	CB 2E	15
SRA (IX+x)	DD CB xx 2E	23
SRA (IY+x)	FD CB xx 2E	23
SRL A	CB 3F	8
SRL B	CB 38	8
SRL C	CB 39	8
SRL D	CB 3A	8
SRL E	CB 3B	8
SRL H	CB 3C	8
SRL L	CB 3D	8
SRL (HL)	CB 3E	15
SRL (IX+x)	DD CB xx 3E	23
SRL (IY+x)	FD CB xx 3E	23
SUB A	97	4
SUB B	90	4
SUB C	91	4
SUB D	92	4
SUB E	93	4
SUB H	94	4
SUB L	95	4
SUB (HL)	96	7
SUB (IX+x)	DD 96 xx	19
SUB (IY+x)	FD 96 xx	19
XOR A	AF	4
XOR B	A8	4
XOR C	A9	4
XOR D	AA	4
XOR E	AB	4
XOR H	AC	4
XOR L	AD	4

Sub A,data D6

XOR data	EE dd	7	XOR (IX+x)	DD AE xx	19
XOR (HL)	AE	7	XOR (IY+x)	FD AE xx	19

xx= index register displacement value (2's complement).
ds= relative jump value (2's complement).
dd= data byte.
al= low byte of address.
ah= high byte of address.
dl= low byte of 16-bit data.
dh= high byte of 16-bit data.
pt= port number.

Note: the clock cycles are given for comparison purposes only, as the clock speed of the Amstrad is affected by the video generating process.

APPENDIX 2

Hexadecimal, decimal and two's complement numbers

Hex	Dec	Dec 2's comp.	Hex	Dec	Dec 2's comp.	Hex	Dec	Dec 2's comp.	Hex	Dec	Dec 2's comp.
00	0	+0	20	32	+32	40	64	+64	60	96	+96
01	1	+1	21	33	+33	41	65	+65	61	97	+97
02	2	+2	22	34	+34	42	66	+66	62	98	+98
03	3	+3	23	35	+35	43	67	+67	63	99	+99
04	4	+4	24	36	+36	44	68	+68	64	100	+100
05	5	+5	25	37	+37	45	69	+69	65	101	+101
06	6	+6	26	38	+38	46	70	+70	66	102	+102
07	7	+7	27	39	+39	47	71	+71	67	103	+103
08	8	+8	28	40	+40	48	72	+72	68	104	+104
09	9	+9	29	41	+41	49	73	+73	69	105	+105
0A	10	+10	2A	42	+42	4A	74	+74	6A	106	+106
0B	11	+11	2B	43	+43	4B	75	+75	6B	107	+107
0C	12	+12	2C	44	+44	4C	76	+76	6C	108	+108
0D	13	+13	2D	45	+45	4D	77	+77	6D	109	+109
0E	14	+14	2E	46	+46	4E	78	+78	6E	110	+110
0F	15	+15	2F	47	+47	4F	79	+79	6F	111	+111
10	16	+16	30	48	+48	50	80	+80	70	112	+112
11	17	+17	31	49	+49	51	81	+81	71	113	+113
12	18	+18	32	50	+50	52	82	+82	72	114	+114
13	19	+19	33	51	+51	53	83	+83	73	115	+115
14	20	+20	34	52	+52	54	84	+84	74	116	+116
15	21	+21	35	53	+53	55	85	+85	75	117	+117
16	22	+22	36	54	+54	56	86	+86	76	118	+118
17	23	+23	37	55	+55	57	87	+87	77	119	+119
18	24	+24	38	56	+56	58	88	+88	78	120	+120
19	25	+25	39	57	+57	59	89	+89	79	121	+121
1A	26	+26	3A	58	+58	5A	90	+90	7A	122	+122
1B	27	+27	3B	59	+59	5B	91	+91	7B	123	+123
1C	28	+28	3C	60	+60	5C	92	+92	7C	124	+124
1D	29	+29	3D	61	+61	5D	93	+93	7D	125	+125
1E	30	+30	3E	62	+62	5E	94	+94	7E	126	+126
1F	31	+31	3F	63	+63	5F	95	+95	7F	127	+127

Hex	Dec	Dec 2's comp.	Hex	Dec	Dec 2's comp.	Hex	Dec	Dec 2's comp.	Hex	Dec	Dec 2's comp.
80	128	-128	A0	160	-96	C0	192	-64	E0	224	-32
81	129	-127	A1	161	-95	C1	193	-63	E1	225	-31
82	130	-126	A2	162	-94	C2	194	-62	E2	226	-30
83	131	-125	A3	163	-93	C3	195	-61	E3	227	-29
84	132	-124	A4	164	-92	C4	196	-60	E4	228	-28
85	133	-123	A5	165	-91	C5	197	-59	E5	229	-27
86	134	-122	A6	166	-90	C6	198	-58	E6	230	-26
87	135	-121	A7	167	-89	C7	199	-57	E7	231	-25
88	136	-120	A8	168	-88	C8	200	-56	E8	232	-24
89	137	-119	A9	169	-87	C9	201	-55	E9	233	-23
8A	138	-118	AA	170	-86	CA	202	-54	EA	234	-22
8B	139	-117	AB	171	-85	CB	203	-53	EB	235	-21
8C	140	-116	AC	172	-84	CC	204	-52	EC	236	-20
8D	141	-115	AD	173	-83	CD	205	-51	ED	237	-19
8E	142	-114	AE	174	-82	CE	206	-50	EE	238	-18
8F	143	-113	AF	175	-81	CF	207	-49	EF	239	-17
90	144	-112	B0	176	-80	D0	208	-48	F0	240	-16
91	145	-111	B1	177	-79	D1	209	-47	F1	241	-15
92	146	-110	B2	178	-78	D2	210	-46	F2	242	-14
93	147	-109	B3	179	-77	D3	211	-45	F3	243	-13
94	148	-108	B4	180	-76	D4	212	-44	F4	244	-12
95	149	-107	B5	181	-75	D5	213	-43	F5	245	-11
96	150	-106	B6	182	-74	D6	214	-42	F6	246	-10
97	151	-105	B7	183	-73	D7	215	-41	F7	247	-9
98	152	-104	B8	184	-72	D8	216	-40	F8	248	-8
99	153	-103	B9	185	-71	D9	217	-39	F9	249	-7
9A	154	-102	BA	186	-70	DA	218	-38	FA	250	-6
9B	155	-101	BB	187	-69	DB	219	-37	FB	251	-5
9C	156	-100	BC	188	-68	DC	220	-36	FC	252	-4
9D	157	-99	BD	189	-67	DD	221	-35	FD	253	-3
9E	158	-98	BE	190	-66	DE	222	-34	FE	254	-2
9F	159	-97	BF	191	-65	DF	223	-33	FF	255	-1

Index

Accumulator (A register) 10
ADC 10
ADD 10
Address bus 1, 3
Alternative register set 4, 5, 28
AND 11, 49, 122
Arithmetic 10
Arithmetic and logic unit 3, 4
Array descriptor blocks 68
Assembler programs 23
Assembly 21
Auxiliary carry flag 5, 11, 13
AY–3–8912 26
Base co-ordinates 108
BASIC ROM 25
Binary coded decimal (BCD) 5, 13
Binary numbers 1, 98
Binary searching 101
Bit 12
Bits 1
Block moves 14
Bubble sort 34
Bytes 1
CALL 9, 17
CALL (BASIC) 40
Caps lock simulation 87
Carry flag 4, 11, 41, 51, 56
Cassette firmware routines 30
CCF 11
Circle formula 91
Clear 5
Clock 3, 26, 42
Code loading routine 39
Colours 109
Comparisons 79
Complement 11
Conditional calls 10
Conditional jumps 9
Conditional returns 88, 125
Control codes 23, 128
Control unit 3, 4
Corruption 20, 125
Counters 78
CP 12, 21
CPD 15
CPI 15
CPIR 15
CPL 11
CPU 25
CRT chip 26
DAA 13
Data bus 1, 3
Delayed replacement sort 43
Delays 126
Devpac 23
DI 16
Direct addressing 6
Disassembly 23
Disk firmware routines 31
Displacements 5
DJNZ 9, 17, 20
Documentation 18, 125
Dynamic memory 5
EI 16
8-bit multiply 98
8255 chip 26

8080 CPU 4
Error reporting 124
EX DE,HL 48
Exchange instructions 8
Exponent 44
False 5
Firmware ROM 25
Firmware routines 27
Flags 4, 9
Floating point numbers 43, 44
Flow diagrams 18
Garbage collection 40, 87
Gate array 25
General purpose registers 4
Graphics firmware routines 99, 102
Halt 16
Hexadecimal numbering 2
I/O instructions 15
I/O pin 13
Immediate addressing 6
IN 14
INC 10
Index registers 5, 77, 84
Indexed addressing 24
Ink encoding 112
INPUT 18
Instructions 6, Appendix 1
Integer variables 37, 40
Interrupt modes 16
Interrupt vector register 5
Interrupts 16, 28
IX register 5
IY register 5
Joysticks 27, 29
Jumpblock 27
Kernel firmware routines 31, 122
Keyboard 18, 27
Keyboard firmware routines 28, 88
KM WAIT CHAR 20
Labels 20
Latching 26
LDD 14
LDDR 15
LDI 14
LDIR 14, 81
Loading instructions 6
Logic 5
Logical instructions 11
Loops 9, 17, 36
Machine firmware routines 31
Mantissa 44
Manual assembly 21
Masking 111
Memory pointers 5, 6, 14, 20
Messages 127
Mnemonics 6
Monitor programs 23
Multiplication 78, 98
NEG 11
NOP 16, 122
Object code 21, 23
Offsets 5, 22
Operation codes (op codes) 6
OR 11
Out 14
Overflow 4
Overflow flag 5
Palette registers 109
Parameter passing 40
Parity/overflow flag 5, 15
Performing instructions 6
Physical co-ordinates 108
Pixel layout 108
Plotting circles 91
POP 8
Ports 26
PPI chip 26
Print co-ordinates 107
Printer routine 32
Printing strings 127
Program counter 3, 6, 9
Program jumps 8
PSG chip 26
PUSH 8, 42

RAM 2, 25
Random events 6
Real numbers 43
Recursion 129
Redefining keys 87
Refresh register 5
Register pairs 4
Registers 3
Reiteration 17
Relative graphic co-ordinates 108
Relative jumps 9, 21, 70
Relocatable code 70
Reset 5, 8
Resident system extensions 119
Restarts 15, 27
Restoring the SP 100
RET 17
RETN 16
Return address 128
Returning data 20
RL 12
RLC 12
RLD 13
ROM 2, 25, 27
Rotate 12, 51
RR 12
RRA 12
RRC 12
RRCA 12
RRD 13
RST *See restarts*
RSX 119
RSX tables 122
SBC 10, 41
Screen addresses 110
Screen base address 110
Screen co-ordinates 107
Screen firmware routines 29, 17
Screen layout 108
Screen modes 109
Screen offset 111
Self-modifying code 24
Sending data 20
SET 5, 8
Set carry flag (SCF) 8
Shell-Metzner sort 43, 101
Shift 12
Shifting bits 98
Sign flag 5
Signed numbers 2, 40
6845 chip 26
16-bit subtraction 50
SLA 12
Sound 31
Source code 23
Square roots 100
SRA 13
SRL 12
Stack operations 8
Stack pointer 4, 7, 8
Standard graphics co-ords 108
Storing code in strings 87
String comparisons 57
String descriptors 53
String printing 124
Strings 53
SUB 10
Subroutines 16, 124
Subtract flag 4
Testing for digits 69
Testing for zero 41
Testing values 12
Text firmware routines 29
The code machine 23
True 5
Two's complement 2, 7, 9, 50, Appendix 2
TXT OUTPUT 20
Upper case characters 85, 87
User graphics co-ordinates 108
Variables 20, 24
Vectoring 27
Video display 26
Video flyback 31

Video RAM 25, 108
Waiting 127
Wild cards 73
XOR 11, 49, 51, 70, 117
Z80 1, 3
Zero flag 5

Other titles from Sunshine

SPECTRUM BOOKS

ZX Spectrum Astronomy
Maurice Gavin
ISBN 0 946408 24 6
£6.95

Spectrum Adventures
A guide to playing and writing adventures
Tony Bridge & Roy Carnell
ISBN 0 946408 07 6
£5.95

Spectrum Machine Code Applications
David Laine
ISBN 0 946408 17 3
£6.95

The Working Spectrum
David Lawrence
ISBN 0 946408 00 9
£5.95

Master your ZX Microdrive
Andrew Pennell
ISBN 0 946408 19 X
£6.95

COMMODORE 64 BOOKS

Mathematics for the Commodore 64
Czes Kosniowski
ISBN 0 946408 14 9
£5.95

Advanced Programming Techniques on the Commodore 64
David Lawrence
ISBN 0 946408 23 8
£5.95

Graphic Art for the Commodore 64
Boris Allan
ISBN 0 946408 15 7
£5.95

Commodore 64 Adventures
Mike Grace
ISBN 0 946408 11 4
£5.95

Business Applications for the Commodore 64
James Hall
ISBN 0 946408 12 2
£5.95

The Working Commodore 64
David Lawrence
ISBN 0 946408 02 5
£5.95

Commodore 64 Machine Code Master
David Lawrence & Mark England
ISBN 0 946408 05 X
£6.95

ELECTRON BOOKS

Graphic Art for the Electron
Boris Allan
ISBN 0 946408 20 3
£5.95

Programming for Education on the Electron Computer
John Scriven & Patrick Hall
ISBN 0 946408 21 1
£5.95

BBC COMPUTER BOOKS

Functional Forth for the BBC computer
Boris Allan
ISBN 0 946408 04 1
£5.95

Graphic Art for the BBC computer
Boris Allan
ISBN 0 946408 08 4
£5.95

DIY Robotics and Sensors for the BBC computer
John Billingsley
ISBN 0 946408 13 0
£6.95

Programming for Education on the BBC computer
John Scriven & Patrick Hall
ISBN 0 946408 10 6
£5.95

The Amstrad CPC 464 has become established as a reliable, easy to use, low cost home micro, and now it has been joined by the equally appealing 664. In this book Jeff Naylor and Diane Rogers show you how to make better use of your Amstrad, by programming in machine code. This makes programs run much faster, and opens the door on a host of new applications for the CPC464 and 664

The book describes the reasons for wanting to tackle machine code programming, before giving a detailed overview of the hardware of the Amstrad. There are routines to show how to write a database program, how to boost the graphics, how to produce graphs, and how to write programs without using the BASIC language at all.

Jeff Naylor and Diane Rogers write with the person who is new to these concepts very much in mind, so their book will be ideal for newcomers to machine code who would like to expand their use of the computer to encompass machine code.

The Authors

Jeff Naylor and Diane Rogers are the authors of **Inside your Spectrum**, which was described by **What Micro** as essential reading for anyone wanting to get into Spectrum machine code programming. Jeff Naylor is a regular contributor to **Popular Computing Weekly** and other micro magazines

GB £ NET +006.95

ISBN 0-946408-80-7

00695

9 780946 408801

SUNSHINE

£6.95 net

ISBN 0 946408 80 7